GNU Emacs Manual

Sixth Edition, Emacs Version 18

for Unix Users

February 1988
(General Public License upgraded, January 1991)

Richard Stallman

Preface

This manual documents the use and simple customization of the Emacs editor. The reader is not expected to be a programmer. Even simple customizations do not require programming skill, but the user who is not interested in customizing can ignore the scattered customization hints.

This is primarily a reference manual, but can also be used as a primer. However, I recommend that the newcomer first use the on-line, learn-by-doing tutorial, which you get by running Emacs and typing `C-h t`. With it, you learn Emacs by using Emacs on a specially designed file which describes commands, tells you when to try them, and then explains the results you see. This gives a more vivid introduction than a printed manual.

On first reading, just skim chapters one and two, which describe the notational conventions of the manual and the general appearance of the Emacs display screen. Note which questions are answered in these chapters, so you can refer back later. After reading chapter four you should practice the commands there. The next few chapters describe fundamental techniques and concepts that are used constantly. You need to understand them thoroughly, experimenting with them if necessary.

To find the documentation on a particular command, look in the index. Keys (character commands) and command names have separate indexes. There is also a glossary, with a cross reference for each term.

This manual comes in two forms: the published form and the Info form. The Info form is for on-line perusal with the INFO program; it is distributed along with GNU Emacs. Both forms contain substantially the same text and are generated from a common source file, which is also distributed along with GNU Emacs.

GNU Emacs is a member of the Emacs editor family. There are many Emacs editors, all sharing common principles of organization. For information on the underlying philosophy of Emacs and the lessons learned from its development, write for a copy of AI memo 519a, "Emacs, the Extensible, Customizable Self-Documenting Display Editor", to Publications Department, Artificial Intelligence Lab, 545 Tech Square, Cambridge, MA 02139, USA. At last report they charge $2.25 per copy. Another useful publication is LCS TM-165, "A Cookbook for an Emacs", by Craig Finseth, available from Publications Department, Laboratory for Computer Science, 545 Tech Square, Cambridge, MA 02139, USA. The price today is $3.

This edition of the manual is intended for use with GNU Emacs installed on Unix systems. GNU Emacs can also be used on VMS systems, which have different file name syntax and do not support all GNU Emacs features. A VMS edition of this manual may appear in the future.

Distribution

GNU Emacs is *free*; this means that everyone is free to use it and free to redistribute it on a free basis. GNU Emacs is not in the public domain; it is copyrighted and there are restrictions on its distribution, but these restrictions are designed to permit everything that a good cooperating citizen would want to do. What is not allowed is to try to prevent others from further sharing any version of GNU Emacs that they might get from you. The precise conditions are found in the GNU General Public License that comes with Emacs and also appears following this section.

The easiest way to get a copy of GNU Emacs is from someone else who has it. You need not ask for permission to do so, or tell any one else; just copy it.

If you have access to the Internet, you can get the latest distribution version of GNU Emacs from host '`prep.ai.mit.edu`' using anonymous login. See the file '`/u2/emacs/GETTING.GNU.SOFTWARE`' on that host to find out about your options for copying and which files to use.

You may also receive GNU Emacs when you buy a computer. Computer manufacturers are free to distribute copies on the same terms that apply to everyone else. These terms require them to give you the full sources, including whatever changes they may have made, and to permit you to redistribute the GNU Emacs received from them under the usual terms of the General Public License. In other words, the program must be free for you when you get it, not just free for the manufacturer.

If you cannot get a copy in any of those ways, you can order one from the Free Software Foundation. Though Emacs itself is free, our distribution service is not. An order form is included at the end of manuals printed by the Foundation. It is also included in the file '`etc/DISTRIB`' in the Emacs distribution. For further information, write to

> Free Software Foundation
> 675 Mass Ave
> Cambridge, MA 02139
> USA

The income from distribution fees goes to support the foundation's purpose: the development of more free software to distribute just like GNU Emacs.

If you find GNU Emacs useful, please **send a donation** to the Free Software Foundation. This will help support development of the rest of the GNU system, and other useful software beyond that. Your donation is tax deductible.

GNU GENERAL PUBLIC LICENSE

Version 1, February 1989

Preamble

The license agreements of most software companies try to keep users at the mercy of those companies. By contrast, our General Public License is intended to guarantee your freedom to share and change free software—to make sure the software is free for all its users. The General Public License applies to the Free Software Foundation's software and to any other program whose authors commit to using it. You can use it for your programs, too.

When we speak of free software, we are referring to freedom, not price. Specifically, the General Public License is designed to make sure that you have the freedom to give away or sell copies of free software, that you receive source code or can get it if you want it, that you can change the software or use pieces of it in new free programs; and that you know you can do these things.

To protect your rights, we need to make restrictions that forbid anyone to deny you these rights or to ask you to surrender the rights. These restrictions translate to certain responsibilities for you if you distribute copies of the software, or if you modify it.

For example, if you distribute copies of a such a program, whether gratis or for a fee, you must give the recipients all the rights that you have. You must make sure that they, too, receive or can get the source code. And you must tell them their rights.

We protect your rights with two steps: (1) copyright the software, and (2) offer you this license which gives you legal permission to copy, distribute and/or modify the software.

Also, for each author's protection and ours, we want to make certain that everyone understands that there is no warranty for this free software. If the software is modified by someone else and passed on, we want its recipients to know that what they have is not the original, so that any problems introduced by others will not reflect on the original authors' reputations.

The precise terms and conditions for copying, distribution and modification follow.

TERMS AND CONDITIONS

1. This License Agreement applies to any program or other work which contains a notice placed by the copyright holder saying it may be distributed under the terms of this General Public License. The "Program", below, refers to any such program or work, and a "work based on the Program" means either the Program or any work containing the Program or a portion of it, either verbatim or with modifications. Each licensee is addressed as "you".
2. You may copy and distribute verbatim copies of the Program's source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice and disclaimer of warranty; keep intact all the notices that refer to this General Public License and to the absence of any warranty; and give any other recipients of the Program a copy of this General Public License along with the Program. You may charge a fee for the physical act of transferring a copy.
3. You may modify your copy or copies of the Program or any portion of it, and copy and distribute such modifications under the terms of Paragraph 1 above, provided that you also do the following:
   - cause the modified files to carry prominent notices stating that you changed the files and the date of any change; and
   - cause the whole of any work that you distribute or publish, that in whole or in part contains the Program or any part thereof, either with or without modifications, to be licensed at no charge to all third parties under the terms of this General Public License (except that you may choose to grant warranty protection to some or all third parties, at your option).
   - If the modified program normally reads commands interactively when run, you must cause it, when started running for such interactive use in the simplest and most usual way, to print or display an announcement including an appropriate copyright notice and a notice that there is no warranty (or else, saying that you provide a warranty) and that users may redistribute the program under these conditions, and telling the user how to view a copy of this General Public License.
   - You may charge a fee for the physical act of transferring a copy, and you may at your option offer warranty protection in exchange for a fee.

   Mere aggregation of another independent work with the Program (or its derivative) on a volume of a storage or distribution medium does not bring the other work under the scope of these terms.
4. You may copy and distribute the Program (or a portion or derivative of it, under Paragraph

2) in object code or executable form under the terms of Paragraphs 1 and 2 above provided that you also do one of the following:

- accompany it with the complete corresponding machine-readable source code, which must be distributed under the terms of Paragraphs 1 and 2 above; or,
- accompany it with a written offer, valid for at least three years, to give any third party free (except for a nominal charge for the cost of distribution) a complete machine-readable copy of the corresponding source code, to be distributed under the terms of Paragraphs 1 and 2 above; or,
- accompany it with the information you received as to where the corresponding source code may be obtained. (This alternative is allowed only for noncommercial distribution and only if you received the program in object code or executable form alone.)

Source code for a work means the preferred form of the work for making modifications to it. For an executable file, complete source code means all the source code for all modules it contains; but, as a special exception, it need not include source code for modules which are standard libraries that accompany the operating system on which the executable file runs, or for standard header files or definitions files that accompany that operating system.

5. You may not copy, modify, sublicense, distribute or transfer the Program except as expressly provided under this General Public License. Any attempt otherwise to copy, modify, sublicense, distribute or transfer the Program is void, and will automatically terminate your rights to use the Program under this License. However, parties who have received copies, or rights to use copies, from you under this General Public License will not have their licenses terminated so long as such parties remain in full compliance.
6. By copying, distributing or modifying the Program (or any work based on the Program) you indicate your acceptance of this license to do so, and all its terms and conditions.
7. Each time you redistribute the Program (or any work based on the Program), the recipient automatically receives a license from the original licensor to copy, distribute or modify the Program subject to these terms and conditions. You may not impose any further restrictions on the recipients' exercise of the rights granted herein.
8. The Free Software Foundation may publish revised and/or new versions of the General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

   Each version is given a distinguishing version number. If the Program specifies a version number of the license which applies to it and "any later version", you have the option of following the terms and conditions either of that version or of any later version published by the Free Software Foundation. If the Program does not specify a version number of the license, you may choose any version ever published by the Free Software Foundation.
9. If you wish to incorporate parts of the Program into other free programs whose distribution conditions are different, write to the author to ask for permission. For software which is copyrighted by the Free Software Foundation, write to the Free Software Foundation; we

sometimes make exceptions for this. Our decision will be guided by the two goals of preserving the free status of all derivatives of our free software and of promoting the sharing and reuse of software generally.

NO WARRANTY

10. BECAUSE THE PROGRAM IS LICENSED FREE OF CHARGE, THERE IS NO WARRANTY FOR THE PROGRAM, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE PROGRAM "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE PROGRAM IS WITH YOU. SHOULD THE PROGRAM PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.
11. IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MAY MODIFY AND/OR REDISTRIBUTE THE PROGRAM AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE PROGRAM (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE PROGRAM TO OPERATE WITH ANY OTHER PROGRAMS), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

END OF TERMS AND CONDITIONS

Appendix: How to Apply These Terms to Your New Programs

If you develop a new program, and you want it to be of the greatest possible use to humanity, the best way to achieve this is to make it free software which everyone can redistribute and change under these terms.

To do so, attach the following notices to the program. It is safest to attach them to the start of each source file to most effectively convey the exclusion of warranty; and each file should have at least the "copyright" line and a pointer to where the full notice is found.

```
one line to give the program's name and a brief idea of what it does.
Copyright (C) 19yy  name of author

This program is free software; you can redistribute it and/or modify
it under the terms of the GNU General Public License as published by
the Free Software Foundation; either version 1, or (at your option)
any later version.

This program is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
GNU General Public License for more details.

You should have received a copy of the GNU General Public License
along with this program; if not, write to the Free Software
Foundation, Inc., 675 Mass Ave, Cambridge, MA 02139, USA.
```

Also add information on how to contact you by electronic and paper mail.

If the program is interactive, make it output a short notice like this when it starts in an interactive mode:

```
Gnomovision version 69, Copyright (C) 19yy name of author
Gnomovision comes with ABSOLUTELY NO WARRANTY; for details type ‘show w’.
This is free software, and you are welcome to redistribute it
under certain conditions; type ‘show c’ for details.
```

The hypothetical commands 'show w' and 'show c' should show the appropriate parts of the General Public License. Of course, the commands you use may be called something other than 'show w' and 'show c'; they could even be mouse-clicks or menu items—whatever suits your program.

You should also get your employer (if you work as a programmer) or your school, if any, to sign a "copyright disclaimer" for the program, if necessary. Here a sample; alter the names:

```
Yoyodyne, Inc., hereby disclaims all copyright interest in the
program 'Gnomovision' (a program to direct compilers to make passes
at assemblers) written by James Hacker.

signature of Ty Coon, 1 April 1989
Ty Coon, President of Vice
```

That's all there is to it!

Introduction

You are reading about GNU Emacs, the GNU incarnation of the advanced, self-documenting, customizable, extensible real-time display editor Emacs. (The ‘G’ in ‘GNU’ is not silent.)

We say that Emacs is a *display* editor because normally the text being edited is visible on the screen and is updated automatically as you type your commands. See Chapter 1 [Screen], page 13.

We call it a *real-time* editor because the display is updated very frequently, usually after each character or pair of characters you type. This minimizes the amount of information you must keep in your head as you edit. See Section 4.5 [Basic Editing], page 28.

We call Emacs advanced because it provides facilities that go beyond simple insertion and deletion: filling of text; automatic indentation of programs; viewing two or more files at once; and dealing in terms of characters, words, lines, sentences, paragraphs, and pages, as well as expressions and comments in several different programming languages. It is much easier to type one command meaning “go to the end of the paragraph” than to find that spot with simple cursor keys.

Self-documenting means that at any time you can type a special character, `Control-h`, to find out what your options are. You can also use it to find out what any command does, or to find all the commands that pertain to a topic. See Section 8.4 [Help], page 48.

Customizable means that you can change the definitions of Emacs commands in little ways. For example, if you use a programming language in which comments start with ‘`<**`’ and end with ‘`**>`’, you can tell the Emacs comment manipulation commands to use those strings (see Section 21.6.2 [Comments], page 152). Another sort of customization is rearrangement of the command set. For example, if you prefer the four basic cursor motion commands (up, down, left and right) on keys in a diamond pattern on the keyboard, you can have it. See Chapter 28 [Customization], page 217.

Extensible means that you can go beyond simple customization and write entirely new commands, programs in the Lisp language to be run by Emacs’s own Lisp interpreter. Emacs is an “on-line extensible” system, which means that it is divided into many functions that call each other, any of which can be redefined in the middle of an editing session. Any part of Emacs can be replaced without making a separate copy of all of Emacs. Most of the editing commands of Emacs are written in Lisp already; the few exceptions could have been written in Lisp but are written in C for efficiency. Although only a programmer can write an extension, anybody can use it afterward.

1 The Organization of the Screen

Emacs divides the screen into several areas, each of which contains its own sorts of information. The biggest area, of course, is the one in which you usually see the text you are editing.

When you are using Emacs, the screen is divided into a number of *windows*. Initially there is one text window occupying all but the last line, plus the special *echo area* or *minibuffer window* in the last line. The text window can be subdivided horizontally or vertically into multiple text windows, each of which can be used for a different file (see Chapter 17 [Windows], page 111). The window that the cursor is in is the *selected window*, in which editing takes place. The other windows are just for reference unless you select one of them.

Each text window's last line is a *mode line* which describes what is going on in that window. It is in inverse video if the terminal supports that, and contains text that starts like '`-----Emacs:` *something*'. Its purpose is to indicate what buffer is being displayed above it in the window; what major and minor modes are in use; and whether the buffer's text has been changed.

1.1 Point

When Emacs is running, the terminal's cursor shows the location at which editing commands will take effect. This location is called *point*. Other commands move point through the text, so that you can edit at different places in it.

While the cursor appears to point *at* a character, point should be thought of as *between* two characters; it points *before* the character that the cursor appears on top of. Sometimes people speak of "the cursor" when they mean "point", or speak of commands that move point as "cursor motion" commands.

Terminals have only one cursor, and when output is in progress it must appear where the typing is being done. This does not mean that point is moving. It is only that Emacs has no way to show you the location of point except when the terminal is idle.

If you are editing several files in Emacs, each file has its own point location. A file that is not being displayed remembers where point is so that it can be seen when you look at that file again.

When there are multiple text windows, each window has its own point location. The cursor shows the location of point in the selected window. This also is how you can tell which window is selected. If the same buffer appears in more than one window, point can be moved in each window independently.

The term 'point' comes from the character '`.`', which was the command in TECO (the language in which the original Emacs was written) for accessing the value now called 'point'.

1.2 The Echo Area

The line at the bottom of the screen (below the mode line) is the *echo area*. It is used to display small amounts of text for several purposes.

Echoing means printing out the characters that you type. Emacs never echoes single-character commands, and multi-character commands are echoed only if you pause while typing them. As soon as you pause for more than a second in the middle of a command, all the characters of the command so far are echoed. This is intended to *prompt* you for the rest of the command. Once echoing has started, the rest of the command is echoed immediately when you type it. This behavior is designed to give confident users fast response, while giving hesitant users maximum feedback. You can change this behavior by setting a variable (see Section 12.4 [Display Vars], page 67).

If a command cannot be executed, it may print an *error message* in the echo area. Error messages are accompanied by a beep or by flashing the screen. Also, any input you have typed ahead is thrown away when an error happens.

Some commands print informative messages in the echo area. These messages look much like error messages, but they are not announced with a beep and do not throw away input. Sometimes the message tells you what the command has done, when this is not obvious from looking at the text being edited. Sometimes the sole purpose of a command is to print a message giving you specific information. For example, the command `C-x =` is used to print a message describing the character position of point in the text and its current column in the window. Commands that take a long time often display messages ending in '`...`' while they are working, and add '`done`' at the end when they are finished.

The echo area is also used to display the *minibuffer*, a window that is used for reading arguments to commands, such as the name of a file to be edited. When the minibuffer is in use, the echo area begins with a prompt string that usually ends with a colon; also, the cursor appears in that line because it is the selected window. You can always get out of the minibuffer by typing `C-g`. See Chapter 6 [Minibuffer], page 35.

1.3 The Mode Line

Each text window's last line is a *mode line* which describes what is going on in that window. When there is only one text window, the mode line appears right above the echo area. The mode line is in inverse video if the terminal supports that, starts and ends with dashes, and contains text like '`Emacs:` *something*'.

If a mode line has something else in place of '`Emacs:` *something*', then the window above it is in a special subsystem such as Dired. The mode line then indicates the status of the subsystem.

Normally, the mode line has the following appearance:

`--`*ch*`-Emacs:` *buf*   `(`*major minor*`)----`*pos*`------`

This gives information about the buffer being displayed in the window: the buffer's name, what major and minor modes are in use, whether the buffer's text has been changed, and how far down the buffer you are currently looking.

ch contains two stars '`**`' if the text in the buffer has been edited (the buffer is "modified"), or '`--`' if the buffer has not been edited. Exception: for a read-only buffer, it is '`%%`'.

buf is the name of the window's chosen *buffer*. The chosen buffer in the selected window (the window that the cursor is in) is also Emacs's selected buffer, the one that editing takes place in. When we speak of what some command does to "the buffer", we are talking about the currently selected buffer. See Chapter 16 [Buffers], page 105.

pos tells you whether there is additional text above the top of the screen, or below the bottom. If your file is small and it is all on the screen, *pos* is '`All`'. Otherwise, it is '`Top`' if you are looking at the beginning of the file, '`Bot`' if you are looking at the end of the file, or '*nn*`%`', where *nn* is the percentage of the file above the top of the screen.

major is the name of the *major mode* in effect in the buffer. At any time, each buffer is in one and only one of the possible major modes. The major modes available include Fundamental mode (the least specialized), Text mode, Lisp mode, and C mode. See Chapter 18 [Major Modes], page 115, for details of how the modes differ and how to select one.

minor is a list of some of the *minor modes* that are turned on at the moment in the window's chosen buffer. '`Fill`' means that Auto Fill mode is on. '`Abbrev`' means that Word Abbrev mode is on. '`Ovwrt`' means that Overwrite mode is on. See Section 28.1 [Minor Modes], page 217, for more information. '`Narrow`' means that the buffer being displayed has editing restricted to only a portion of its text. This is not really a minor mode, but is like one. See Section 27.2 [Narrowing], page 208. '`Def`' means that a keyboard macro is being defined. See Section 28.3 [Keyboard Macros], page 223.

Some buffers display additional information after the minor modes. For example, Rmail buffers display the current message number and the total number of messages. Compilation buffers and Shell mode display the status of the subprocess.

In addition, if Emacs is currently inside a recursive editing level, square brackets ('`[...]`') appear around the parentheses that surround the modes. If Emacs is in one recursive editing level within another, double square brackets appear, and so on. Since this information pertains to Emacs in general and not to any one buffer, the square brackets appear in every mode line on the screen or not in any of them. See Section 27.1 [Recursive Edit], page 207.

Emacs can optionally display the time and system load in all mode lines. To enable this feature, type `M-x display-time`. The information added to the mode line usually appears after the file name, before the mode names and their parentheses. It looks like this:

    hh:*mm*pm *l.ll* [*d*]

(Some fields may be missing if your operating system cannot support them.) *hh* and *mm* are the hour and minute, followed always by '`am`' or '`pm`'. *l.ll* is the average number of running processes in the whole system recently. *d* is an approximate index of the ratio of disk activity to cpu activity for all users.

The word '`Mail`' appears after the load level if there is mail for you that you have not read yet.

Customization note: the variable `mode-line-inverse-video` controls whether the mode line is displayed in inverse video (assuming the terminal supports it); `nil` means no inverse video. The default is `t`.

2 Characters, Keys and Commands

This chapter explains the character set used by Emacs for input commands and for the contents of files, and also explains the concepts of *keys* and *commands* which are necessary for understanding how your keyboard input is understood by Emacs.

2.1 The Emacs Character Set

GNU Emacs uses the ASCII character set, which defines 128 different character codes. Some of these codes are assigned graphic symbols such as 'a' and '='; the rest are control characters, such as `Control-a` (also called `C-a` for short). `C-a` gets its name from the fact that you type it by holding down the `CTRL` key and then pressing `a`. There is no distinction between `C-a` and `C-A`; they are the same character.

Some control characters have special names, and special keys you can type them with: `RET`, `TAB`, `LFD`, `DEL` and `ESC`. The space character is usually referred to below as `SPC`, even though strictly speaking it is a graphic character whose graphic happens to be blank.

Emacs extends the 7-bit ASCII code to an 8-bit code by adding an extra bit to each character. This makes 256 possible command characters. The additional bit is called Meta. Any ASCII character can be made Meta; examples of Meta characters include `Meta-a` (`M-a`, for short), `M-A` (not the same character as `M-a`, but those two characters normally have the same meaning in Emacs), `M-RET`, and `M-C-a`. For traditional reasons, `M-C-a` is usually called `C-M-a`; logically speaking, the order in which the modifier keys `CTRL` and `META` are mentioned does not matter.

Some terminals have a `META` key, and allow you to type Meta characters by holding this key down. Thus, `Meta-a` is typed by holding down `META` and pressing `a`. The `META` key works much like the `SHIFT` key. Such a key is not always labeled `META`, however, as this function is often a special option for a key with some other primary purpose.

If there is no `META` key, you can still type Meta characters using two-character sequences starting with `ESC`. Thus, to enter `M-a`, you could type `ESC a`. To enter `C-M-a`, you would type `ESC C-a`. `ESC` is allowed on terminals with Meta keys, too, in case you have formed a habit of using it.

Emacs believes the terminal has a `META` key if the variable `meta-flag` is non-`nil`. Normally this is set automatically according to the termcap entry for your terminal type. However, sometimes the termcap entry is wrong, and then it is useful to set this variable yourself. See Section 28.2 [Variables], page 218, for how to do this.

Emacs buffers also use an 8-bit character set, because bytes have 8 bits, but only the ASCII characters are considered meaningful. ASCII graphic characters in Emacs buffers are displayed with their graphics. `LFD` is the same as a newline character; it is displayed by starting a new line. `TAB` is displayed by moving to the next tab stop column (usually every 8 columns). Other control characters are displayed as a caret (‘`^`’) followed by the non-control version of the character; thus, `C-a` is displayed as ‘`^A`’. Non-ASCII characters 128 and up are displayed with octal escape sequences; thus, character code 243 (octal), also called `M-#` when used as an input character, is displayed as ‘`\243`’.

2.2 Keys

A *complete key*—where ‘key’ is short for *key sequence*—is a sequence of keystrokes that are understood by Emacs as a unit, as a single command (possibly undefined). Most single characters constitute complete keys in the standard Emacs command set; there are also some multi-character keys. Examples of complete keys are `C-a`, `X`, `RET`, `C-x C-f` and `C-x 4 C-f`.

A *prefix key* is a sequence of keystrokes that are the beginning of a complete key, but not a whole one. Prefix keys and complete keys are collectively called *keys*.

A prefix key is the beginning of a series of longer sequences that are valid keys; adding any single character to the end of the prefix gives a valid key, which could be defined as an Emacs command, or could be a prefix itself. For example, `C-x` is standardly defined as a prefix, so `C-x` and the next input character combine to make a two-character key. There are 256 different two-character keys starting with `C-x`, one for each possible second character. Many of these two-character keys starting with `C-x` are standardly defined as Emacs commands. Notable examples include `C-x C-f` and `C-x s` (see Chapter 15 [Files], page 87).

Adding one character to a prefix key does not have to form a complete key. It could make another, longer prefix. For example, `C-x 4` is itself a prefix that leads to 256 different three-character keys, including `C-x 4 f`, `C-x 4 b` and so on. It would be possible to define one of those three-character sequences as a prefix, creating a series of four-character keys, but we did not define any of them this way.

By contrast, the two-character sequence `C-f C-k` is not a key, because the `C-f` is a complete key in itself. It’s impossible to give `C-f C-k` an independent meaning as a command as long as `C-f` retains its meaning. `C-f C-k` is two commands.

All told, the prefix keys in Emacs are `C-c`, `C-x`, `C-h`, `C-x 4`, and `ESC`. But this is not built in; it is just a matter of Emacs's standard key bindings. In customizing Emacs, you could make new prefix keys, or eliminate these. See Section 28.4 [Key Bindings], page 226.

Whether a sequence is a key can be changed by customization. For example, if you redefine `C-f` as a prefix, `C-f C-k` automatically becomes a key (complete, unless you define it too as a prefix). Conversely, if you remove the prefix definition of `C-x 4`, then `C-x 4 f` (or `C-x 4` *anything*) is no longer a key.

2.3 Keys and Commands

This manual is full of passages that tell you what particular keys do. But Emacs does not assign meanings to keys directly. Instead, Emacs assigns meanings to *functions*, and then gives keys their meanings by *binding* them to functions.

A function is a Lisp object that can be executed as a program. Usually it is a Lisp symbol which has been given a function definition; every symbol has a name, usually made of a few English words separated by dashes, such as `next-line` or `forward-word`. It also has a *definition* which is a Lisp program; this is what makes the function do what it does. Only some functions can be the bindings of keys; these are functions whose definitions use `interactive` to specify how to call them interactively. Such functions are called *commands*, and their names are *command names*. More information on this subject will appear in the *GNU Emacs Lisp Manual* (which is not yet written).

The bindings between keys and functions are recorded in various tables called *keymaps*. See Section 28.4.1 [Keymaps], page 226.

When we say that "`C-n` moves down vertically one line" we are glossing over a distinction that is irrelevant in ordinary use but is vital in understanding how to customize Emacs. It is the function `next-line` that is programmed to move down vertically. `C-n` has this effect *because* it is bound to that function. If you rebind `C-n` to the function `forward-word` then `C-n` will move forward by words instead. Rebinding keys is a common method of customization.

In the rest of this manual, we usually ignore this subtlety to keep things simple. To give the customizer the information he needs, we state the name of the command which really does the work in parentheses after mentioning the key that runs it. For example, we will say that "The command

`C-n` (`next-line`) moves point vertically down," meaning that `next-line` is a command that moves vertically down and `C-n` is a key that is standardly bound to it.

While we are on the subject of information for customization only, it's a good time to tell you about *variables*. Often the description of a command will say, "To change this, set the variable `mumble-foo`." A variable is a name used to remember a value. Most of the variables documented in this manual exist just to facilitate customization: some command or other part of Emacs examines the variable and behaves differently accordingly. Until you are interested in customizing, you can ignore the information about variables. When you are ready to be interested, read the basic information on variables, and then the information on individual variables will make sense. See Section 28.2 [Variables], page 218.

3 Entering and Exiting Emacs

The usual way to invoke Emacs is just to type `emacs RET` at the shell. Emacs clears the screen and then displays an initial advisor message and copyright notice. You can begin typing Emacs commands immediately afterward.

Some operating systems insist on discarding all type-ahead when Emacs starts up; they give Emacs no way to prevent this. Therefore, it is wise to wait until Emacs clears the screen before typing your first editing command.

Before Emacs reads the first command, you have not had a chance to give a command to specify a file to edit. But Emacs must always have a current buffer for editing. In an attempt to do something useful, Emacs presents a buffer named '`*scratch*`' which is in Lisp Interaction mode; you can use it to type Lisp expressions and evaluate them, or you can ignore that capability and simply doodle. (You can specify a different major mode for this buffer by setting the variable `initial-major-mode` in your init file. See Section 28.6 [Init File], page 232.)

It is also possible to specify files to be visited, Lisp files to be loaded, and functions to be called, by giving Emacs arguments in the shell command line. See Section 3.2 [Command Switches], page 22.

3.1 Exiting Emacs

There are two commands for exiting Emacs because there are two kinds of exiting: *suspending* Emacs and *killing* Emacs. *Suspending* means stopping Emacs temporarily and returning control to its superior (usually the shell), allowing you to resume editing later in the same Emacs job, with the same files, same kill ring, same undo history, and so on. This is the usual way to exit. *Killing* Emacs means destroying the Emacs job. You can run Emacs again later, but you will get a fresh Emacs; there is no way to resume the same editing session after it has been killed.

`C-z` Suspend Emacs (`suspend-emacs`).

`C-x C-c` Kill Emacs (`save-buffers-kill-emacs`).

To suspend Emacs, type `C-z` (`suspend-emacs`). This takes you back to the shell from which you invoked Emacs. You can resume Emacs with the command `%emacs` if you are using the C shell.

On systems that do not permit programs to be suspended, `C-z` runs an inferior shell that communicates directly with the terminal, and Emacs waits until you exit the subshell. The only way on these systems to get back to the shell from which Emacs was run (to log out, for example) is to kill Emacs. `C-d` or `exit` are typical commands to exit a subshell.

To kill Emacs, type `C-x C-c` (`save-buffers-kill-emacs`). A two-character key is used for this to make it harder to type. Unless a numeric argument is used, this command first offers to save any modified buffers. If you do not save them all, it asks for reconfirmation with `yes` before killing Emacs, since any changes not saved before that will be lost forever. Also, if any subprocesses are still running, `C-x C-c` asks for confirmation about them, since killing Emacs will kill the subprocesses immediately.

In most programs running on Unix, certain characters may instantly suspend or kill the program. (In Berkeley Unix these characters are normally `C-z` and `C-c`.) **This Unix feature is turned off while you are in Emacs.** The meanings of `C-z` and `C-x C-c` as keys in Emacs were inspired by the standard Berkeley Unix meanings of `C-z` and `C-c`, but that is their only relationship with Unix. You could customize these keys to do anything (see Section 28.4.1 [Keymaps], page 226).

3.2 Command Line Switches and Arguments

GNU Emacs supports command line arguments to request various actions when invoking Emacs. These are for compatibility with other editors and for sophisticated activities. They are not needed for ordinary editing with Emacs, so new users can skip this section.

You may be used to using command line arguments with other editors to specify which file to edit. That's because many other editors are designed to be started afresh each time you want to edit. You edit one file and then exit the editor. The next time you want to edit either another file or the same one, you must run the editor again. With these editors, it makes sense to use a command line argument to say which file to edit.

The recommended way to use GNU Emacs is to start it only once, just after you log in, and do all your editing in the same Emacs process. Each time you want to edit a different file, you visit it with the existing Emacs, which eventually comes to have many files in it ready for editing. Usually you do not kill the Emacs until you are about to log out.

When files are nearly always read by typing commands to an editor that is already running, command line arguments for specifying a file when the editor is started are seldom needed.

Emacs accepts command-line arguments that specify files to visit, functions to call, and other activities and operating modes.

The command arguments are processed in the order they appear in the command argument list; however, certain arguments (the ones in the second table) must be at the front of the list if they are used.

Here are the arguments allowed:

'*file*'
: Visit *file* using `find-file`. See Section 15.2 [Visiting], page 88.

'`+`*linenum file*'
: Visit *file* using `find-file`, then go to line number *linenum* in it.

'`-l` *file*'
'`-load` *file*'
: Load a file *file* of Lisp code with the function `load`. See Section 22.3 [Lisp Libraries], page 169.

'`-f` *function*'
'`-funcall` *function*'
: Call Lisp function *function* with no arguments.

'`-i` *file*'
'`-insert` *file*'
: Insert the contents of *file* into the current buffer. This is like what `M-x insert-buffer` does; See Section 15.8 [Misc File Ops], page 102.

'`-kill`'
: Exit from Emacs without asking for confirmation.

The remaining switches are recognized only at the beginning of the command line. If more than one of them appears, they must appear in the order that they appear in this table.

'`-t` *device*'
: Use *device* as the device for terminal input and output.

'`-d` *display*'
: When running with the X window system, use the display named *display* to make the window that serves as Emacs's terminal.

'`-batch`'
: Run Emacs in *batch mode*, which means that the text being edited is not displayed and the standard Unix interrupt characters such as `C-z` and `C-c` continue to have their normal effect. Emacs in batch mode outputs to `stdout` only what would normally be printed in the echo area under program control.

  Batch mode is used for running programs written in Emacs Lisp from shell scripts, makefiles, and so on. Normally the '`-l`' switch or '`-f`' switch will be used as well, to invoke a Lisp program to do the batch processing.

'`-batch`' implies '`-q`' (do not load an init file). It also causes Emacs to kill itself after all command switches have been processed. In addition, auto-saving is not done except in buffers for which it has been explicitly requested.

'`-q`'
'`-no-init-file`'
: Do not load your Emacs init file '`~/.emacs`'.

'`-u` *user*'
'`-user` *user*'
: Load *user*'s Emacs init file '`~`*user*`/.emacs`' instead of your own.

Note that the init file can get access to the command line argument values as the elements of a list in the variable `command-line-args`. (The arguments in the second table above will already have been processed and will not be in the list.) The init file can override the normal processing of the other arguments by setting this variable.

One way to use command switches is to visit many files automatically:

```
emacs *.c
```

passes each `.c` file as a separate argument to Emacs, so that Emacs visits each file (see Section 15.2 [Visiting], page 88).

Here is an advanced example that assumes you have a Lisp program file called '`hack-c-program.el`' which, when loaded, performs some useful operation on current buffer, expected to be a C program.

```
emacs -batch foo.c -l hack-c-program -f save-buffer -kill > log
```

Here Emacs is told to visit '`foo.c`', load '`hack-c-program.el`' (which makes changes in the visited file), save '`foo.c`' (note that `save-buffer` is the function that `C-x C-s` is bound to), and then exit to the shell that this command was done with. '`-batch`' guarantees there will be no problem redirecting output to '`log`', because Emacs will not assume that it has a display terminal to work with.

4 Basic Editing Commands

We now give the basics of how to enter text, make corrections, and save the text in a file. If this material is new to you, you might learn it more easily by running the Emacs learn-by-doing tutorial. To do this, type `Control-h t` (`help-with-tutorial`).

4.1 Inserting Text

To insert printing characters into the text you are editing, just type them. This inserts the character into the buffer at the cursor (that is, at *point*; see Section 1.1 [Point], page 13). The cursor moves forward. Any characters after the cursor move forward too. If the text in the buffer is '`FOOBAR`', with the cursor before the '`B`', then if you type `XX`, you get '`FOOXXBAR`', with the cursor still before the '`B`'.

To *delete* text you have just inserted, use `DEL`. `DEL` deletes the character *before* the cursor (not the one that the cursor is on top of or under; that is the character *after* the cursor). The cursor and all characters after it move backwards. Therefore, if you type a printing character and then type `DEL`, they cancel out.

To end a line and start typing a new one, type `RET`. This inserts a newline character in the buffer. If point is in the middle of a line, `RET` splits the line. Typing `DEL` when the cursor is at the beginning of a line rubs out the newline before the line, thus joining the line with the preceding line.

Emacs will split lines automatically when they become too long, if you turn on a special mode called *Auto Fill* mode. See Section 20.6 [Filling], page 134, for how to use Auto Fill mode.

Customization information: `DEL` in most modes runs the command named `delete-backward-char`; `RET` runs the command `newline`, and self-inserting printing characters run the command `self-insert`, which inserts whatever character was typed to invoke it. Some major modes rebind `DEL` to other commands.

Direct insertion works for printing characters and `SPC`, but other characters act as editing commands and do not insert themselves. If you need to insert a control character or a character whose code is above 200 octal, you must *quote* it by typing the character `control-q` (`quoted-insert`) first. There are two ways to use `C-q`:

- `Control-q` followed by any non-graphic character (even `C-g`) inserts that character.
- `Control-q` followed by three octal digits inserts the character with the specified character code.

A numeric argument to `C-q` specifies how many copies of the quoted character should be inserted (see Section 4.9 [Arguments], page 31).

If you prefer to have text characters replace (overwrite) existing text rather than shove it to the right, you can enable Overwrite mode, a minor mode. See Section 28.1 [Minor Modes], page 217.

4.2 Changing the Location of Point

To do more than insert characters, you have to know how to move point (see Section 1.1 [Point], page 13). Here are a few of the commands for doing that.

`C-a` Move to the beginning of the line (`beginning-of-line`).

`C-e` Move to the end of the line (`end-of-line`).

`C-f` Move forward one character (`forward-char`).

`C-b` Move backward one character (`backward-char`).

`M-f` Move forward one word (`forward-word`).

`M-b` Move backward one word (`backward-word`).

`C-n` Move down one line, vertically (`next-line`). This command attempts to keep the horizontal position unchanged, so if you start in the middle of one line, you end in the middle of the next. When on the last line of text, `C-n` creates a new line and moves onto it.

`C-p` Move up one line, vertically (`previous-line`).

`C-l` Clear the screen and reprint everything (`recenter`). Text moves on the screen to bring point to the center of the window.

`M-r` Move point to left margin on the line halfway down the screen or window (`move-to-window-line`). Text does not move on the screen. A numeric argument says how many screen lines down from the top of the window (zero for the top). A negative argument counts from the bottom (-1 for the bottom).

`C-t` Transpose two characters, the ones before and after the cursor (`transpose-chars`).

`M-<` Move to the top of the buffer (`beginning-of-buffer`). With numeric argument n, move to $n/10$ of the way from the top. See Section 4.9 [Arguments], page 31, for more information on numeric arguments.

`M->` Move to the end of the buffer (`end-of-buffer`).

`M-x goto-char`
: Read a number *n* and move cursor to character number *n*. Position 1 is the beginning of the buffer.

`M-x goto-line`
: Read a number *n* and move cursor to line number *n*. Line 1 is the beginning of the buffer.

`C-x C-n`
: Use the current column of point as the *semipermanent goal column* for `C-n` and `C-p` (`set-goal-column`). Henceforth, those commands always move to this column in each line moved into, or as close as possible given the contents of the line. This goal column remains in effect until canceled.

`C-u C-x C-n`
: Cancel the goal column. Henceforth, `C-n` and `C-p` once again try to avoid changing the horizontal position, as usual.

If you set the variable `track-eol` to a non-`nil` value, then `C-n` and `C-p` when at the end of the starting line move to the end of the line. Normally, `track-eol` is `nil`.

4.3 Erasing Text

`DEL` Delete the character before the cursor (`delete-backward-char`).

`C-d` Delete the character after the cursor (`delete-char`).

`C-k` Kill to the end of the line (`kill-line`).

`M-d` Kill forward to the end of the next word (`kill-word`).

`M-DEL` Kill back to the beginning of the previous word (`backward-kill-word`).

You already know about the `DEL` key which deletes the character before the cursor. Another key, `Control-d`, deletes the character after the cursor, causing the rest of the text on the line to shift left. If `Control-d` is typed at the end of a line, that line and the next line are joined together.

To erase a larger amount of text, use the `Control-k` key, which kills a line at a time. If `C-k` is done at the beginning or middle of a line, it kills all the text up to the end of the line. If `C-k` is done at the end of a line, it joins that line and the next line.

See Section 10.1.3 [Killing], page 55, for more flexible ways of killing text.

4.4 Files

The commands above are sufficient for creating and altering text in an Emacs buffer; the more advanced Emacs commands just make things easier. But to keep any text permanently you must put it in a *file*. Files are named units of text which are stored by the operating system for you to retrieve later by name. To look at or use the contents of a file in any way, including editing the file with Emacs, you must specify the file name.

Consider a file named '`/usr/rms/foo.c`'. In Emacs, to begin editing this file, type

```
C-x C-f /usr/rms/foo.c RET
```

Here the file name is given as an *argument* to the command `C-x C-f` (`find-file`). That command uses the *minibuffer* to read the argument, and you type `RET` to terminate the argument (see Chapter 6 [Minibuffer], page 35).

Emacs obeys the command by *visiting* the file: creating a buffer, copying the contents of the file into the buffer, and then displaying the buffer for you to edit. You can make changes in it, and then *save* the file by typing `C-x C-s` (`save-buffer`). This makes the changes permanent by copying the altered contents of the buffer back into the file '`/usr/rms/foo.c`'. Until then, the changes are only inside your Emacs, and the file '`foo.c`' is not changed.

To create a file, just visit the file with `C-x C-f` as if it already existed. Emacs will make an empty buffer in which you can insert the text you want to put in the file. When you save your text with `C-x C-s`, the file will be created.

Of course, there is a lot more to learn about using files. See Chapter 15 [Files], page 87.

4.5 Help

If you forget what a key does, you can find out with the Help character, which is `C-h`. Type `C-h k` followed by the key you want to know about; for example, `C-h k C-n` tells you all about what `C-n` does. `C-h` is a prefix key; `C-h k` is just one of its subcommands (the command `describe-key`). The other subcommands of `C-h` provide different kinds of help. Type `C-h` three times to get a description of all the help facilities. See Section 8.4 [Help], page 48.

4.6 Blank Lines

Here are special commands and techniques for putting in and taking out blank lines.

`C-o` Insert one or more blank lines after the cursor (`open-line`).

`C-x C-o` Delete all but one of many consecutive blank lines (`delete-blank-lines`).

When you want to insert a new line of text before an existing line, you can do it by typing the new line of text, followed by `RET`. However, it may be easier to see what you are doing if you first make a blank line and then insert the desired text into it. This is easy to do using the key `C-o` (`open-line`), which inserts a newline after point but leaves point in front of the newline. After `C-o`, type the text for the new line. `C-o F O O` has the same effect as `F O O RET`, except for the final location of point.

You can make several blank lines by typing `C-o` several times, or by giving it an argument to tell it how many blank lines to make. See Section 4.9 [Arguments], page 31, for how.

If you have many blank lines in a row and want to get rid of them, use `C-x C-o` (`delete-blank-lines`). When point is on a blank line which is adjacent to at least one other blank line, `C-x C-o` deletes all but one of the consecutive blank lines, leaving exactly one. With point on a blank line with no other blank line adjacent to it, the sole blank line is deleted, leaving none. When point is on a nonblank line, `C-x C-o` deletes any blank lines following that nonblank line.

4.7 Continuation Lines

If you add too many characters to one line, without breaking it with a `RET`, the line will grow to occupy two (or more) lines on the screen, with a '`\`' at the extreme right margin of all but the last of them. The '`\`' says that the following screen line is not really a distinct line in the text, but just the *continuation* of a line too long to fit the screen. Sometimes it is nice to have Emacs insert newlines automatically when a line gets too long; for this, use Auto Fill mode (see Section 20.6 [Filling], page 134).

Instead of continuation, long lines can be displayed by *truncation*. This means that all the characters that do not fit in the width of the screen or window do not appear at all. They remain in the buffer, temporarily invisible. '`$`' is used in the last column instead of '`\`' to inform you that truncation is in effect.

Continuation can be turned off for a particular buffer by setting the variable `truncate-lines` to non-`nil` in that buffer. Truncation instead of continuation also happens whenever horizontal scrolling is in use, and optionally whenever side-by-side windows are in use (see Chapter 17 [Windows], page 111). Altering the value of `truncate-lines` makes it local to the current buffer; until that time, the default value is in effect. The default is initially `nil`. See Section 28.2.3 [Locals], page 220.

4.8 Cursor Position Information

If you are accustomed to other display editors, you may be surprised that Emacs does not always display the page number or line number of point in the mode line. This is because the text is stored in a way that makes it difficult to compute this information. Displaying them all the time would be intolerably slow. They are not needed very often in Emacs anyway, but there are commands to compute them and print them.

`M-x what-page`
: Print page number of point, and line number within page.

`M-x what-line`
: Print line number of point in the buffer.

`M-=`
: Print number of lines in the current region (`count-lines-region`).

`C-x =`
: Print character code of character after point, character position of point, and column of point (`what-cursor-position`).

There are two commands for printing line numbers. `M-x what-line` counts lines from the beginning of the file and prints the line number point is on. The first line of the file is line number 1. These numbers can be used as arguments to `M-x goto-line`. By contrast, `M-x what-page` counts pages from the beginning of the file, and counts lines within the page, printing both of them. See Section 20.5 [Pages], page 133.

While on this subject, we might as well mention `M-=` (`count-lines-region`), which prints the number of lines in the region (see Chapter 9 [Mark], page 49). See Section 20.5 [Pages], page 133, for the command `C-x l` which counts the lines in the current page.

The command `C-x =` (`what-cursor-position`) can be used to find out the column that the cursor is in, and other miscellaneous information about point. It prints a line in the echo area that looks like this:

```
Char: x (0170)  point=65986 of 563027(12%)  x=44
```

(In fact, this is the output produced when point is before the '`x=44`' in the example.)

The two values after '`Char:`' describe the character following point, first by showing it and second by giving its octal character code.

'`point=`' is followed by the position of point expressed as a character count. The front of the buffer counts as position 1, one character later as 2, and so on. The next, larger number is the total number of characters in the buffer. Afterward in parentheses comes the position expressed as a percentage of the total size.

'`x=`' is followed by the horizontal position of point, in columns from the left edge of the window.

If the buffer has been narrowed, making some of the text at the beginning and the end temporarily invisible, `C-x =` prints additional text describing the current visible range. For example, it might say

```
Char: x (0170)  point=65986 of 563025(12%) <65102 - 68533>  x=44
```

where the two extra numbers give the smallest and largest character position that point is allowed to assume. The characters between those two positions are the visible ones. See Section 27.2 [Narrowing], page 208.

If point is at the end of the buffer (or the end of the visible part), `C-x =` omits any description of the character after point. The output looks like

```
point=563026 of 563025(100%)  x=0
```

4.9 Numeric Arguments

Any Emacs command can be given a *numeric argument*. Some commands interpret the argument as a repetition count. For example, giving an argument of ten to the key `C-f` (the command `forward-char`, move forward one character) moves forward ten characters. With these commands, no argument is equivalent to an argument of one. Negative arguments are allowed. Often they tell a command to move or act backwards.

If your terminal keyboard has a `META` key, the easiest way to specify a numeric argument is to type digits and/or a minus sign while holding down the the `META` key. For example,

```
M-5 C-n
```

would move down five lines. The characters `Meta-1`, `Meta-2`, and so on, as well as `Meta--`, do this because they are keys bound to commands (`digit-argument` and `negative-argument`) that are defined to contribute to an argument for the next command.

Another way of specifying an argument is to use the `C-u` (`universal-argument`) command followed by the digits of the argument. With `C-u`, you can type the argument digits without holding down shift keys. To type a negative argument, start with a minus sign. Just a minus sign normally means −1. `C-u` works on all terminals.

`C-u` followed by a character which is neither a digit nor a minus sign has the special meaning of "multiply by four". It multiplies the argument for the next command by four. `C-u` twice multiplies it by sixteen. Thus, `C-u C-u C-f` moves forward sixteen characters. This is a good way to move forward "fast", since it moves about 1/5 of a line in the usual size screen. Other useful combinations are `C-u C-n`, `C-u C-u C-n` (move down a good fraction of a screen), `C-u C-u C-o` (make "a lot" of blank lines), and `C-u C-k` (kill four lines).

Some commands care only about whether there is an argument, and not about its value. For example, the command `M-q` (`fill-paragraph`) with no argument fills text; with an argument, it justifies the text as well. (See Section 20.6 [Filling], page 134, for more information on `M-q`.) Just `C-u` is a handy way of providing an argument for such commands.

Some commands use the value of the argument as a repeat count, but do something peculiar when there is no argument. For example, the command `C-k` (`kill-line`) with argument *n* kills *n* lines, including their terminating newlines. But `C-k` with no argument is special: it kills the text up to the next newline, or, if point is right at the end of the line, it kills the newline itself. Thus, two `C-k` commands with no arguments can kill a nonblank line, just like `C-k` with an argument of one. (See Section 10.1.3 [Killing], page 55, for more information on `C-k`.)

A few commands treat a plain `C-u` differently from an ordinary argument. A few others may treat an argument of just a minus sign differently from an argument of −1. These unusual cases will be described when they come up; they are always for reasons of convenience of use of the individual command.

5 Undoing Changes

Emacs allows all changes made in the text of a buffer to be undone, up to a certain amount of change (8000 characters). Each buffer records changes individually, and the undo command always applies to the current buffer. Usually each editing command makes a separate entry in the undo records, but some commands such as `query-replace` make many entries, and very simple commands such as self-inserting characters are often grouped to make undoing less tedious.

`C-x u` Undo one batch of changes (usually, one command worth) (`undo`).

`C-_` The same.

The command `C-x u` or `C-_` is how you undo. The first time you give this command, it undoes the last change. Point moves to the text affected by the undo, so you can see what was undone.

Consecutive repetitions of the `C-_` or `C-x u` commands undo earlier and earlier changes, back to the limit of what has been recorded. If all recorded changes have already been undone, the undo command prints an error message and does nothing.

Any command other than an undo command breaks the sequence of undo commands. Starting at this moment, the previous undo commands are considered ordinary changes that can themselves be undone. Thus, you can redo changes you have undone by typing `C-f` or any other command that will have no important effect, and then using more undo commands.

If you notice that a buffer has been modified accidentally, the easiest way to recover is to type `C-_` repeatedly until the stars disappear from the front of the mode line. At this time, all the modifications you made have been cancelled. If you do not remember whether you changed the buffer deliberately, type `C-_` once, and when you see the last change you made undone, you will remember why you made it. If it was an accident, leave it undone. If it was deliberate, redo the change as described in the preceding paragraph.

Whenever an undo command makes the stars disappear from the mode line, it means that the buffer contents are the same as they were when the file was last read in or saved.

Not all buffers record undo information. Buffers whose names start with spaces don't; these buffers are used internally by Emacs and its extensions to hold text that users don't normally look at or edit. Also, minibuffers, help buffers and documentation buffers don't record undo information.

At most 8000 or so characters of deleted or modified text can be remembered in any one buffer for reinsertion by the undo command. Also, there is a limit on the number of individual insert, delete or change actions that can be remembered.

The reason the `undo` command has two keys, `C-x u` and `C-_`, set up to run it is that it is worthy of a single-character key, but the way to type `C-_` on some keyboards is not obvious. `C-x u` is an alternative you can type in the same fashion on any terminal.

6 The Minibuffer

The *minibuffer* is the facility used by Emacs commands to read arguments more complicated than a single number. Minibuffer arguments can be file names, buffer names, Lisp function names, Emacs command names, Lisp expressions, and many other things, depending on the command reading the argument. The usual Emacs editing commands can be used in the minibuffer to edit the argument.

When the minibuffer is in use, it appears in the echo area, and the terminal's cursor moves there. The beginning of the minibuffer line displays a *prompt* which says what kind of input you should supply and how it will be used. Often this prompt is derived from the name of the command that the argument is for. The prompt normally ends with a colon.

Sometimes a *default argument* appears in parentheses after the colon; it too is part of the prompt. The default will be used as the argument value if you enter an empty argument (e.g., just type `RET`). For example, commands that read buffer names always show a default, which is the name of the buffer that will be used if you type just `RET`.

The simplest way to give a minibuffer argument is to type the text you want, terminated by `RET` which exits the minibuffer. You can get out of the minibuffer, canceling the command that it was for, by typing `C-g`.

Since the minibuffer uses the screen space of the echo area, it can conflict with other ways Emacs customarily uses the echo area. Here is how Emacs handles such conflicts:

- If a command gets an error while you are in the minibuffer, this does not cancel the minibuffer. However, the echo area is needed for the error message and therefore the minibuffer itself is hidden for a while. It comes back after a few seconds, or as soon as you type anything.
- If in the minibuffer you use a command whose purpose is to print a message in the echo area, such as `C-x =`, the message is printed normally, and the minibuffer is hidden for a while. It comes back after a few seconds, or as soon as you type anything.
- Echoing of keystrokes does not take place while the minibuffer is in use.

6.1 Minibuffers for File Names

Sometimes the minibuffer starts out with text in it. For example, when you are supposed to give a file name, the minibuffer starts out containing the *default directory*, which ends with a slash.

This is to inform you which directory the file will be found in if you do not specify a directory. For example, the minibuffer might start out with

```
Find File: /u2/emacs/src/
```

where '`Find File: `' is the prompt. Typing `buffer.c` specifies the file '`/u2/emacs/src/buffer.c`'. To find files in nearby directories, use `..`; thus, if you type `../lisp/simple.el`, the file that you visit will be the one named '`/u2/emacs/lisp/simple.el`'. Alternatively, you can kill with `M-DEL` the directory names you don't want (see Section 20.2 [Words], page 130).

You can also type an absolute file name, one starting with a slash or a tilde, ignoring the default directory. For example, to find the file '`/etc/termcap`', just type the name, giving

```
Find File: /u2/emacs/src//etc/termcap
```

Two slashes in a row are not normally meaningful in Unix file names, but they are allowed in GNU Emacs. They mean, "ignore everything before the second slash in the pair." Thus, '`/u2/emacs/src/`' is ignored, and you get the file '`/etc/termcap`'.

If you set `insert-default-directory` to `nil`, the default directory is not inserted in the minibuffer. This way, the minibuffer starts out empty. But the name you type, if relative, is still interpreted with respect to the same default directory.

6.2 Editing in the Minibuffer

The minibuffer is an Emacs buffer (albeit a peculiar one), and the usual Emacs commands are available for editing the text of an argument you are entering.

Since `RET` in the minibuffer is defined to exit the minibuffer, inserting a newline into the minibuffer must be done with `C-o` or with `C-q LFD`. (Recall that a newline is really the `LFD` character.)

The minibuffer has its own window which always has space on the screen but acts as if it were not there when the minibuffer is not in use. When the minibuffer is in use, its window is just like the others; you can switch to another window with `C-x o`, edit text in other windows and perhaps even visit more files, before returning to the minibuffer to submit the argument. You can kill text in another window, return to the minibuffer window, and then yank the text to use it in the argument. See Chapter 17 [Windows], page 111.

There are some restrictions on the use of the minibuffer window, however. You cannot switch buffers in it—the minibuffer and its window are permanently attached. Also, you cannot split or kill the minibuffer window. But you can make it taller in the normal fashion with `C-x ^`.

If while in the minibuffer you issue a command that displays help text of any sort in another window, then that window is identified as the one to scroll if you type `C-M-v` while in the minibuffer. This lasts until you exit the minibuffer. This feature comes into play if a completing minibuffer gives you a list of possible completions.

Recursive use of the minibuffer is supported by Emacs. However, it is easy to do this by accident (because of autorepeating keyboards, for example) and get confused. Therefore, most Emacs commands that use the minibuffer refuse to operate if the minibuffer window is selected. If the minibuffer is active but you have switched to a different window, recursive use of the minibuffer is allowed—if you know enough to try to do this, you probably will not get confused.

If you set the variable `enable-recursive-minibuffers` to be non-`nil`, recursive use of the minibuffer is always allowed.

6.3 Completion

When appropriate, the minibuffer provides a *completion* facility. This means that you type enough of the argument to determine the rest, based on Emacs's knowledge of which arguments make sense, and Emacs visibly fills in the rest, or as much as can be determined from the part you have typed.

When completion is available, certain keys—`TAB`, `RET`, and `SPC`—are redefined to complete an abbreviation present in the minibuffer into a longer string that it stands for, by matching it against a set of *completion alternatives* provided by the command reading the argument. `?` is defined to display a list of possible completions of what you have inserted.

For example, when the minibuffer is being used by `Meta-x` to read the name of a command, it is given a list of all available Emacs command names to complete against. The completion keys match the text in the minibuffer against all the command names, find any additional characters of the name that are implied by the ones already present in the minibuffer, and add those characters to the ones you have given.

Case is normally significant in completion, because it is significant in most of the names that you can complete (buffer names, file names and command names). Thus, '`fo`' will not complete to

'Foo'. When you are completing a name in which case does not matter, case may be ignored for completion's sake if the program said to do so.

6.3.1 Completion Example

A concrete example may help here. If you type `Meta-x au TAB`, the `TAB` looks for alternatives (in this case, command names) that start with '`au`'. There are only two: `auto-fill-mode` and `auto-save-mode`. These are the same as far as `auto-`, so the '`au`' in the minibuffer changes to '`auto-`'.

If you type `TAB` again immediately, there are multiple possibilities for the very next character—it could be '`s`' or '`f`'—so no more characters are added; but a list of all possible completions is displayed in another window.

If you go on to type `f TAB`, this `TAB` sees '`auto-f`'. The only command name starting this way is `auto-fill-mode`, so completion inserts the rest of that. You now have '`auto-fill-mode`' in the minibuffer after typing just `au TAB f TAB`. Note that `TAB` has this effect because in the minibuffer it is bound to the function `minibuffer-complete` when completion is supposed to be done.

6.3.2 Completion Commands

Here is a list of all the completion commands, defined in the minibuffer when completion is available.

`TAB`
: Complete the text in the minibuffer as much as possible (`minibuffer-complete`).

`SPC`
: Complete the text in the minibuffer but don't add or fill out more than one word (`minibuffer-complete-word`).

`RET`
: Submit the text in the minibuffer as the argument, possibly completing first as described below (`minibuffer-complete-and-exit`).

`?`
: Print a list of all possible completions of the text in the minibuffer (`minibuffer-list-completions`).

`SPC` completes much like `TAB`, but never goes beyond the next hyphen or space. If you have '`auto-f`' in the minibuffer and type `SPC`, it finds that the completion is '`auto-fill-mode`', but it stops completing after '`fill-`'. This gives '`auto-fill-`'. Another `SPC` at this point completes all the way to '`auto-fill-mode`'. `SPC` in the minibuffer runs the function `minibuffer-complete-word` when completion is available.

There are three different ways that `RET` can work in completing minibuffers, depending on how the argument will be used.

- *Strict* completion is used when it is meaningless to give any argument except one of the known alternatives. For example, when `C-x k` reads the name of a buffer to kill, it is meaningless to give anything but the name of an existing buffer. In strict completion, `RET` refuses to exit if the text in the minibuffer does not complete to an exact match.
- *Cautious* completion is similar to strict completion, except that `RET` exits only if the text was an exact match already, not needing completion. If the text is not an exact match, `RET` does not exit, but it does complete the text. If it completes to an exact match, a second `RET` will exit.

  Cautious completion is used for reading file names for files that must already exist.
- *Permissive* completion is used when any string whatever is meaningful, and the list of completion alternatives is just a guide. For example, when `C-x C-f` reads the name of a file to visit, any file name is allowed, in case you want to create a file. In permissive completion, `RET` takes the text in the minibuffer exactly as given, without completing it.

The completion commands display a list of all possible completions in a window whenever there is more than one possibility for the very next character. Also, typing `?` explicitly requests such a list. The list of completions counts as help text, so `C-M-v` typed in the minibuffer scrolls the list.

When completion is done on file names, certain file names are usually ignored. The variable `completion-ignored-extensions` contains a list of strings; a file whose name ends in any of those strings is ignored as a possible completion. The standard value of this variable has several elements including `".o"`, `".elc"`, `".dvi"` and `"~"`. The effect is that, for example, '`foo`' can complete to '`foo.c`' even though '`foo.o`' exists as well. If the only possible completions are files that end in "ignored" strings, then they are not ignored.

Normally, a completion command that finds the next character is undetermined automatically displays a list of all possible completions. If the variable `completion-auto-help` is set to `nil`, this does not happen, and you must type `?` to display the possible completions.

6.4 Repeating Minibuffer Commands

Every command that uses the minibuffer at least once is recorded on a special history list, together with the values of the minibuffer arguments, so that you can repeat the command easily. In particular, every use of `Meta-x` is recorded, since `M-x` uses the minibuffer to read the command name.

`C-x ESC`
: Re-execute a recent minibuffer command (`repeat-complex-command`).

`M-p`
: Within `C-x ESC`, move to previous recorded command (`previous-complex-command`).

`M-n`
: Within `C-x ESC`, move to the next (more recent) recorded command (`next-complex-command`).

`M-x list-command-history`
: Display the entire command history, showing all the commands `C-x ESC` can repeat, most recent first.

`C-x ESC` is used to re-execute a recent minibuffer-using command. With no argument, it repeats the last such command. A numeric argument specifies which command to repeat; one means the last one, and larger numbers specify earlier ones.

`C-x ESC` works by turning the previous command into a Lisp expression and then entering a minibuffer initialized with the text for that expression. If you type just `RET`, the command is repeated as before. You can also change the command by editing the Lisp expression. Whatever expression you finally submit is what will be executed. The repeated command is added to the front of the command history unless it is identical to the most recently executed command already there.

Even if you don't understand Lisp syntax, it will probably be obvious which command is displayed for repetition. If you do not change the text, you can be sure it will repeat exactly as before.

Once inside the minibuffer for `C-x ESC`, if the command shown to you is not the one you want to repeat, you can move around the list of previous commands using `M-n` and `M-p`. `M-p` replaces the contents of the minibuffer with the next earlier recorded command, and `M-n` replaces them with the next later command. After finding the desired previous command, you can edit its expression as usual and then resubmit it by typing `RET` as usual. Any editing you have done on the command to be repeated is lost if you use `M-n` or `M-p`.

`M-p` is more useful than `M-n`, since more often you will initially request to repeat the most recent command and then decide to repeat an older one instead. These keys are specially defined within `C-x ESC` to run the commands `previous-complex-command` and `next-complex-command`.

The list of previous minibuffer-using commands is stored as a Lisp list in the variable `command-history`. Each element is a Lisp expression which describes one command and its arguments. Lisp programs can reexecute a command by feeding the corresponding `command-history` element to `eval`.

7 Running Commands by Name

The Emacs commands that are used often or that must be quick to type are bound to keys—short sequences of characters—for convenient use. Other Emacs commands that do not need to be brief are not bound to keys; to run them, you must refer to them by name.

A command name is, by convention, made up of one or more words, separated by hyphens; for example, `auto-fill-mode` or `manual-entry`. The use of English words makes the command name easier to remember than a key made up of obscure characters, even though it is more characters to type. Any command can be run by name, even if it is also runnable by keys.

The way to run a command by name is to start with `M-x`, type the command name, and finish it with `RET`. `M-x` uses the minibuffer to read the command name. `RET` exits the minibuffer and runs the command.

Emacs uses the minibuffer for reading input for many different purposes; on this occasion, the string '`M-x`' is displayed at the beginning of the minibuffer as a *prompt* to remind you that your input should be the name of a command to be run. See Chapter 6 [Minibuffer], page 35, for full information on the features of the minibuffer.

You can use completion to enter the command name. For example, the command `forward-char` can be invoked by name by typing

```
M-x forward-char RET
```

or

```
M-x fo TAB c RET
```

Note that `forward-char` is the same command that you invoke with the key `C-f`. Any command (interactively callable function) defined in Emacs can be called by its name using `M-x` whether or not any keys are bound to it.

If you type `C-g` while the command name is being read, you cancel the `M-x` command and get out of the minibuffer, ending up at top level.

To pass a numeric argument to the command you are invoking with `M-x`, specify the numeric argument before the `M-x`. `M-x` passes the argument along to the function which it calls. The argument value appears in the prompt while the command name is being read.

Normally, when describing a command that is run by name, we omit the `RET` that is needed to terminate the name. Thus we might speak of `M-x auto-fill-mode` rather than `M-x auto-fill-mode RET`. We mention the `RET` only when there is a need to emphasize its presence, such as when describing a sequence of input that contains a command name and arguments that follow it.

`M-x` is defined to run the command `execute-extended-command`, which is responsible for reading the name of another command and invoking it.

8 Help

Emacs provides extensive help features which revolve around a single character, `C-h`. `C-h` is a prefix key that is used only for documentation-printing commands. The characters that you can type after `C-h` are called *help options*. One help option is `C-h`; that is how you ask for help about using `C-h`.

`C-h C-h` prints a list of the possible help options, and then asks you to go ahead and type the option. It prompts with a string

```
A, B, C, F, I, K, L, M, N, S, T, V, W, C-c, C-d, C-n, C-w or C-h for more help:
```

and you should type one of those characters.

Typing a third `C-h` displays a description of what the options mean; it still waits for you to type an option. To cancel, type `C-g`.

Here is a summary of the defined help commands.

`C-h a` *string* `RET`
: Display list of commands whose names contain *string* (`command-apropos`).

`C-h b`
: Display a table of all key bindings in effect now; local bindings of the current major mode first, followed by all global bindings (`describe-bindings`).

`C-h c` *key*
: Print the name of the command that *key* runs (`describe-key-briefly`). `c` is for 'character'. For more extensive information on *key*, use `C-h k`.

`C-h f` *function* `RET`
: Display documentation on the Lisp function named *function* (`describe-function`). Note that commands are Lisp functions, so a command name may be used.

`C-h i`
: Run Info, the program for browsing documentation files (`info`). The complete Emacs manual is available on-line in Info.

`C-h k` *key*
: Display name and documentation of the command *key* runs (`describe-key`).

`C-h l`
: Display a description of the last 100 characters you typed (`view-lossage`).

`C-h m`
: Display documentation of the current major mode (`describe-mode`).

`C-h n`
: Display documentation of Emacs changes, most recent first (`view-emacs-news`).

`C-h s`
: Display current contents of the syntax table, plus an explanation of what they mean (`describe-syntax`).

`C-h t`
: Display the Emacs tutorial (`help-with-tutorial`).

`C-h v` *var* `RET`
: Display the documentation of the Lisp variable *var* (`describe-variable`).

`C-h w` *command* `RET`
: Print which keys run the command named *command* (`where-is`).

8.1 Documentation for a Key

The most basic `C-h` options are `C-h c` (`describe-key-briefly`) and `C-h k` (`describe-key`). `C-h c` *key* prints in the echo area the name of the command that *key* is bound to. For example, `C-h c C-f` prints '`forward-char`'. Since command names are chosen to describe what the command does, this is a good way to get a very brief description of what *key* does.

`C-h k` *key* is similar but gives more information. It displays the documentation string of the command *key* is bound to as well as its name. This is too big for the echo area, so a window is used for the display.

8.2 Help by Command or Variable Name

`C-h f` (`describe-function`) reads the name of a Lisp function using the minibuffer, then displays that function's documentation string in a window. Since commands are Lisp functions, you can use this to get the documentation of a command that is known by name. For example,

```
C-h f auto-fill-mode RET
```

displays the documentation of `auto-fill-mode`. This is the only way to see the documentation of a command that is not bound to any key (one which you would normally call using `M-x`).

`C-h f` is also useful for Lisp functions that you are planning to use in a Lisp program. For example, if you have just written the code `(make-vector len)` and want to be sure that you are using `make-vector` properly, type `C-h f make-vector RET`. Because `C-h f` allows all function names, not just command names, you may find that some of your favorite abbreviations that work in `M-x` don't work in `C-h f`. An abbreviation may be unique among command names yet fail to be unique when other function names are allowed.

The function name for `C-h f` to describe has a default which is used if you type `RET` leaving the minibuffer empty. The default is the function called by the innermost Lisp expression in the buffer around point, *provided* that is a valid, defined Lisp function name. For example, if point is located following the text '(make-vector (car x)', the innermost list containing point is the one that starts with '(make-vector', so the default is to describe the function make-vector.

C-h f is often useful just to verify that you have the right spelling for the function name. If C-h f mentions a default in the prompt, you have typed the name of a defined Lisp function. If that tells you what you want to know, just type C-g to cancel the C-h f command and go on editing.

C-h w *command* RET tells you what keys are bound to *command*. It prints a list of the keys in the echo area. Alternatively, it says that the command is not on any keys, which implies that you must use M-x to call it.

C-h v (describe-variable) is like C-h f but describes Lisp variables instead of Lisp functions. Its default is the Lisp symbol around or before point, but only if that is the name of a known Lisp variable. See Section 28.2 [Variables], page 218.

8.3 Apropos

A more sophisticated sort of question to ask is, "What are the commands for working with files?" For this, type C-h a file RET, which displays a list of all command names that contain 'file', such as copy-file, find-file, and so on. With each command name appears a brief description of how to use the command, and what keys you can currently invoke it with. For example, it would say that you can invoke find-file by typing C-x C-f. The a in C-h a stands for 'Apropos'; C-h a runs the Lisp function command-apropos.

Because C-h a looks only for functions whose names contain the string which you specify, you must use ingenuity in choosing the string. If you are looking for commands for killing backwards and C-h a kill-backwards RET doesn't reveal any, don't give up. Try just kill, or just backwards, or just back. Be persistent. Pretend you are playing Adventure. Also note that you can use a regular expression as the argument (see Section 13.5 [Regexps], page 74).

Here is a set of arguments to give to C-h a that covers many classes of Emacs commands, since there are strong conventions for naming the standard Emacs commands. By giving you a feel for the naming conventions, this set should also serve to aid you in developing a technique for picking apropos strings.

> char, line, word, sentence, paragraph, region, page, sexp, list, defun, buffer, screen, window, file, dir, register, mode, beginning, end, forward, backward, next, previous,

up, down, search, goto, kill, delete, mark, insert, yank, fill, indent, case, change, set, what, list, find, view, describe.

To list all Lisp symbols that contain a match for a regexp, not just the ones that are defined as commands, use the command `M-x apropos` instead of `C-h a`.

8.4 Other Help Commands

`C-h i` (`info`) runs the Info program, which is used for browsing through structured documentation files. The entire Emacs manual is available within Info. Eventually all the documentation of the GNU system will be available. Type `h` after entering Info to run a tutorial on using Info.

If something surprising happens, and you are not sure what commands you typed, use `C-h l` (`view-lossage`). `C-h l` prints the last 100 command characters you typed in. If you see commands that you don't know, you can use `C-h c` to find out what they do.

Emacs has several major modes, each of which redefines a few keys and makes a few other changes in how editing works. `C-h m` (`describe-mode`) prints documentation on the current major mode, which normally describes all the commands that are changed in this mode.

`C-h b` (`describe-bindings`) and `C-h s` (`describe-syntax`) present other information about the current Emacs mode. `C-h b` displays a list of all the key bindings now in effect; the local bindings of the current major mode first, followed by the global bindings (see Section 28.4 [Key Bindings], page 226). `C-h s` displays the contents of the syntax table, with explanations of each character's syntax (see Section 28.5 [Syntax], page 229).

The other `C-h` options display various files of useful information. `C-h C-w` displays the full details on the complete absence of warranty for GNU Emacs. `C-h n` (`view-emacs-news`) displays the file '`emacs/etc/NEWS`', which contains documentation on Emacs changes arranged chronologically. `C-h t` (`help-with-tutorial`) displays the learn-by-doing Emacs tutorial. `C-h C-c` (`describe-copying`) displays the file '`emacs/etc/COPYING`', which tells you the conditions you must obey in distributing copies of Emacs. `C-h C-d` (`describe-distribution`) displays another file named '`emacs/etc/DISTRIB`', which tells you how you can order a copy of the latest version of Emacs.

9 The Mark and the Region

There are many Emacs commands which operate on an arbitrary contiguous part of the current buffer. To specify the text for such a command to operate on, you set *the mark* at one end of it, and move point to the other end. The text between point and the mark is called *the region*. You can move point or the mark to adjust the boundaries of the region. It doesn't matter which one is set first chronologically, or which one comes earlier in the text.

Once the mark has been set, it remains until it is set again at another place. The mark remains fixed with respect to the preceding character if text is inserted or deleted in the buffer. Each Emacs buffer has its own mark, so that when you return to a buffer that had been selected previously, it has the same mark it had before.

Many commands that insert text, such as `C-y` (`yank`) and `M-x insert-buffer`, position the mark at one end of the inserted text—the opposite end from where point is positioned, so that the region contains the text just inserted.

Aside from delimiting the region, the mark is also useful for remembering a spot that you may want to go back to. To make this feature more useful, Emacs remembers 16 previous locations of the mark, in the `mark ring`.

9.1 Setting the Mark

Here are some commands for setting the mark:

`C-SPC` Set the mark where point is (`set-mark-command`).

`C-@` The same.

`C-x C-x` Interchange mark and point (`exchange-point-and-mark`).

For example, if you wish to convert part of the buffer to all upper-case, you can use the `C-x C-u` (`upcase-region`) command, which operates on the text in the region. You can first go to the beginning of the text to be capitalized, type `C-SPC` to put the mark there, move to the end, and then type `C-x C-u`. Or, you can set the mark at the end of the text, move to the beginning, and then type `C-x C-u`. Most commands that operate on the text in the region have the word `region` in their names.

The most common way to set the mark is with the `C-SPC` command (`set-mark-command`). This sets the mark where point is. Then you can move point away, leaving the mark behind. It is actually incorrect to speak of the character `C-SPC`; there is no such character. When you type `SPC` while holding down `CTRL`, what you get on most terminals is the character `C-@`. This is the key actually bound to `set-mark-command`. But unless you are unlucky enough to have a terminal where typing `C-SPC` does not produce `C-@`, you might as well think of this character as `C-SPC`.

Since terminals have only one cursor, there is no way for Emacs to show you where the mark is located. You have to remember. The usual solution to this problem is to set the mark and then use it soon, before you forget where it is. But you can see where the mark is with the command `C-x C-x` (`exchange-point-and-mark`) which puts the mark where point was and point where the mark was. The extent of the region is unchanged, but the cursor and point are now at the previous location of the mark.

`C-x C-x` is also useful when you are satisfied with the location of point but want to move the mark; do `C-x C-x` to put point there and then you can move it. A second use of `C-x C-x`, if necessary, puts the mark at the new location with point back at its original location.

9.2 Operating on the Region

Once you have created an active region, you can do many things to the text in it:

- Kill it with `C-w` (see Section 10.1.3 [Killing], page 55).
- Save it in a register with `C-x x` (see Chapter 11 [Registers], page 63).
- Save it in a buffer or a file (see Section 10.3 [Accumulating Text], page 58).
- Convert case with `C-x C-l` or `C-x C-u`
  (see Section 20.7 [Case], page 137).
- Evaluate it as Lisp code with `M-x eval-region` (see Section 22.4 [Lisp Eval], page 172).
- Fill it as text with `M-g` (see Section 20.6 [Filling], page 134).
- Print hardcopy with `M-x print-region` (see Section 27.5 [Hardcopy], page 213).
- Indent it with `C-x TAB` or `C-M-\` (see Chapter 19 [Indentation], page 117).

9.3 Commands to Mark Textual Objects

There are commands for placing point and the mark around a textual object such as a word, list, paragraph or page.

`M-@` Set mark after end of next word (`mark-word`). This command and the following one do not move point.

`C-M-@` Set mark after end of next Lisp expression (`mark-sexp`).

`M-h` Put region around current paragraph (`mark-paragraph`).

`C-M-h` Put region around current Lisp defun (`mark-defun`).

`C-x h` Put region around entire buffer (`mark-whole-buffer`).

`C-x C-p` Put region around current page (`mark-page`).

`M-@` (`mark-word`) puts the mark at the end of the next word, while `C-M-@` (`mark-sexp`) puts it at the end of the next Lisp expression. These characters allow you to save a little typing or redisplay, sometimes.

Other commands set both point and mark, to delimit an object in the buffer. `M-h` (`mark-paragraph`) moves point to the beginning of the paragraph that surrounds or follows point, and puts the mark at the end of that paragraph (see Section 20.4 [Paragraphs], page 132). `M-h` does all that's necessary if you wish to indent, case-convert, or kill a whole paragraph. `C-M-h` (`mark-defun`) similarly puts point before and the mark after the current or following defun (see Section 21.3 [Defuns], page 142). `C-x C-p` (`mark-page`) puts point before the current page (or the next or previous, according to the argument), and mark at the end (see Section 20.5 [Pages], page 133). The mark goes after the terminating page delimiter (to include it), while point goes after the preceding page delimiter (to exclude it). Finally, `C-x h` (`mark-whole-buffer`) sets up the entire buffer as the region, by putting point at the beginning and the mark at the end.

9.4 The Mark Ring

Aside from delimiting the region, the mark is also useful for remembering a spot that you may want to go back to. To make this feature more useful, Emacs remembers 16 previous locations of the mark, in the *mark ring*. Most commands that set the mark push the old mark onto this ring. To return to a marked location, use `C-u C-SPC` (or `C-u C-@`); this is the command `set-mark-command` given a numeric argument. It moves point to where the mark was, and restores the mark from the ring of former marks. So repeated use of this command moves point to all of the old marks on the ring, one by one. The marks you see go to the end of the ring, so no marks are lost.

Each buffer has its own mark ring. All editing commands use the current buffer's mark ring. In particular, `C-u C-SPC` always stays in the same buffer.

Many commands that can move long distances, such as `M-<` (`beginning-of-buffer`), start by setting the mark and saving the old mark on the mark ring. This is to make it easier for you to move back later. Searches do this except when they do not actually move point. You can tell when a command sets the mark because '`Mark Set`' is printed in the echo area.

The variable `mark-ring-max` is the maximum number of entries to keep in the mark ring. If that many entries exist and another one is pushed, the last one in the list is discarded. Repeating `C-u C-SPC` circulates through the limited number of entries that are currently in the ring.

The variable `mark-ring` holds the mark ring itself, as a list of marker objects in the order most recent first. This variable is local in every buffer.

10 Killing and Moving Text

Killing means erasing text and copying it into the *kill ring*, from which it can be retrieved by *yanking* it. Some other systems that have recently become popular use the terms "cutting" and "pasting" for these operations.

The commonest way of moving or copying text with Emacs is to kill it and later yank it in one or more places. This is very safe because all the text killed recently is remembered, and it is versatile, because the many commands for killing syntactic units can also be used for moving those units. There are also other ways of copying text for special purposes.

Emacs has only one kill ring, so you can kill text in one buffer and yank it in another buffer.

10.1 Deletion and Killing

Most commands which erase text from the buffer save it so that you can get it back if you change your mind, or move or copy it to other parts of the buffer. These commands are known as *kill* commands. The rest of the commands that erase text do not save it; they are known as *delete* commands. (This distinction is made only for erasure of text in the buffer.)

The delete commands include `C-d` (`delete-char`) and `DEL` (`delete-backward-char`), which delete only one character at a time, and those commands that delete only spaces or newlines. Commands that can destroy significant amounts of nontrivial data generally kill. The commands' names and individual descriptions use the words '`kill`' and '`delete`' to say which they do. If you do a kill or delete command by mistake, you can use the `C-x u` (`undo`) command to undo it (see Chapter 5 [Undo], page 33).

10.1.1 Deletion

`C-d` Delete next character (`delete-char`).

`DEL` Delete previous character (`delete-backward-char`).

`M-\` Delete spaces and tabs around point (`delete-horizontal-space`).

`M-SPC` Delete spaces and tabs around point, leaving one space (`just-one-space`).

`C-x C-o` Delete blank lines around the current line (`delete-blank-lines`).

`M-^` Join two lines by deleting the intervening newline, and any indentation following it (`delete-indentation`).

The most basic delete commands are `C-d` (`delete-char`) and `DEL` (`delete-backward-char`). `C-d` deletes the character after point, the one the cursor is "on top of". Point doesn't move. `DEL` deletes the character before the cursor, and moves point back. Newlines can be deleted like any other characters in the buffer; deleting a newline joins two lines. Actually, `C-d` and `DEL` aren't always delete commands; if given an argument, they kill instead, since they can erase more than one character this way.

The other delete commands are those which delete only formatting characters: spaces, tabs and newlines. `M-\` (`delete-horizontal-space`) deletes all the spaces and tab characters before and after point. `M-SPC` (`just-one-space`) does likewise but leaves a single space after point, regardless of the number of spaces that existed previously (even zero).

`C-x C-o` (`delete-blank-lines`) deletes all blank lines after the current line, and if the current line is blank deletes all blank lines preceding the current line as well (leaving one blank line, the current line). `M-^` (`delete-indentation`) joins the current line and the previous line, or the current line and the next line if given an argument, by deleting a newline and all surrounding spaces, possibly leaving a single space. See Chapter 19 [Indentation], page 117.

10.1.2 Killing by Lines

`C-k` Kill rest of line or one or more lines (`kill-line`).

The simplest kill command is `C-k`. If given at the beginning of a line, it kills all the text on the line, leaving it blank. If given on a blank line, the blank line disappears. As a consequence, if you go to the front of a non-blank line and type `C-k` twice, the line disappears completely.

More generally, `C-k` kills from point up to the end of the line, unless it is at the end of a line. In that case it kills the newline following the line, thus merging the next line into the current one. Invisible spaces and tabs at the end of the line are ignored when deciding which case applies, so if point appears to be at the end of the line, you can be sure the newline will be killed.

If `C-k` is given a positive argument, it kills that many lines and the newlines that follow them (however, text on the current line before point is spared). With a negative argument, it kills back to a number of line beginnings. An argument of -2 means kill back to the second line beginning.

If point is at the beginning of a line, that line beginning doesn't count, so `C-u - 2 C-k` with point at the front of a line kills the two previous lines.

`C-k` with an argument of zero kills all the text before point on the current line.

10.1.3 Other Kill Commands

`C-w` Kill region (from point to the mark) (`kill-region`). See Section 20.2 [Words], page 130.

`M-d` Kill word (`kill-word`).

`M-DEL` Kill word backwards (`backward-kill-word`).

`C-x DEL` Kill back to beginning of sentence (`backward-kill-sentence`). See Section 20.3 [Sentences], page 131.

`M-k` Kill to end of sentence (`kill-sentence`).

`C-M-k` Kill sexp (`kill-sexp`). See Section 21.2 [Lists], page 140.

`M-z` *char* Kill up to next occurrence of *char* (`zap-to-char`).

A kill command which is very general is `C-w` (`kill-region`), which kills everything between point and the mark. With this command, you can kill any contiguous sequence of characters, if you first set the mark at one end of them and go to the other end.

A convenient way of killing is combined with searching: `M-z` (`zap-to-char`) reads a character and kills from point up to (but not including) the next occurrence of that character in the buffer. If there is no next occurrence, killing goes to the end of the buffer. A numeric argument acts as a repeat count. A negative argument means to search backward and kill text before point.

Other syntactic units can be killed: words, with `M-DEL` and `M-d` (see Section 20.2 [Words], page 130); sexps, with `C-M-k` (see Section 21.2 [Lists], page 140); and sentences, with `C-x DEL` and `M-k` (see Section 20.3 [Sentences], page 131).

10.2 Yanking

Yanking is getting back text which was killed. This is what some systems call "pasting". The usual way to move or copy text is to kill it and then yank it one or more times.

`C-y` Yank last killed text (`yank`).

`M-y` Replace re-inserted killed text with the previously killed text (`yank-pop`).

`M-w` Save region as last killed text without actually killing it (`copy-region-as-kill`).

`C-M-w` Append next kill to last batch of killed text (`append-next-kill`).

10.2.1 The Kill Ring

All killed text is recorded in the *kill ring*, a list of blocks of text that have been killed. There is only one kill ring, used in all buffers, so you can kill text in one buffer and yank it in another buffer. This is the usual way to move text from one file to another. (See Section 10.3 [Accumulating Text], page 58, for some other ways.)

The command `C-y` (`yank`) reinserts the text of the most recent kill. It leaves the cursor at the end of the text. It sets the mark at the beginning of the text. See Chapter 9 [Mark], page 49.

`C-u C-y` leaves the cursor in front of the text, and sets the mark after it. This is only if the argument is specified with just a `C-u`, precisely. Any other sort of argument, including `C-u` and digits, has an effect described below (under "Yanking Earlier Kills").

If you wish to copy a block of text, you might want to use `M-w` (`copy-region-as-kill`), which copies the region into the kill ring without removing it from the buffer. This is approximately equivalent to `C-w` followed by `C-y`, except that `M-w` does not mark the buffer as "modified" and does not temporarily change the screen.

10.2.2 Appending Kills

Normally, each kill command pushes a new block onto the kill ring. However, two or more kill commands in a row combine their text into a single entry, so that a single `C-y` gets it all back as it was before it was killed. This means that you don't have to kill all the text in one command; you can keep killing line after line, or word after word, until you have killed it all, and you can still get it all back at once. (Thus we join television in leading people to kill thoughtlessly.)

Commands that kill forward from point add onto the end of the previous killed text. Commands that kill backward from point add onto the beginning. This way, any sequence of mixed forward and backward kill commands puts all the killed text into one entry without rearrangement. Numeric arguments do not break the sequence of appending kills. For example, suppose the buffer contains

```
This is the first
line of sample text
and here is the third.
```

with point at the beginning of the second line. If you type `C-k C-u 2 M-DEL C-k`, the first `C-k` kills the text '`line of sample text`', `C-u 2 M-DEL` kills '`the first`' with the newline that followed it, and the second `C-k` kills the newline after the second line. The result is that the buffer contains '`This is and here is the third.`' and a single kill entry contains '`the firstRETline of sample textRET`'—all the killed text, in its original order.

If a kill command is separated from the last kill command by other commands (not just numeric arguments), it starts a new entry on the kill ring. But you can force it to append by first typing the command `C-M-w` (`append-next-kill`) in front of it. The `C-M-w` tells the following command, if it is a kill command, to append the text it kills to the last killed text, instead of starting a new entry. With `C-M-w`, you can kill several separated pieces of text and accumulate them to be yanked back in one place.

10.2.3 Yanking Earlier Kills

To recover killed text that is no longer the most recent kill, you need the `Meta-y` (`yank-pop`) command. `M-y` can be used only after a `C-y` or another `M-y`. It takes the text previously yanked and replaces it with the text from an earlier kill. So, to recover the text of the next-to-the-last kill, you first use `C-y` to recover the last kill, and then use `M-y` to replace it with the previous kill.

You can think in terms of a "last yank" pointer which points at an item in the kill ring. Each time you kill, the "last yank" pointer moves to the newly made item at the front of the ring. `C-y` yanks the item which the "last yank" pointer points to. `M-y` moves the "last yank" pointer to a different item, and the text in the buffer changes to match. Enough `M-y` commands can move the pointer to any item in the ring, so you can get any item into the buffer. Eventually the pointer reaches the end of the ring; the next `M-y` moves it to the first item again.

Yanking moves the "last yank" pointer around the ring, but it does not change the order of the entries in the ring, which always runs from the most recent kill at the front to the oldest one still remembered.

`M-y` can take a numeric argument, which tells it how many items to advance the "last yank" pointer by. A negative argument moves the pointer toward the front of the ring; from the front of the ring, it moves to the last entry and starts moving forward from there.

Once the text you are looking for is brought into the buffer, you can stop doing `M-y` commands and it will stay there. It's just a copy of the kill ring item, so editing it in the buffer does not change what's in the ring. As long as no new killing is done, the "last yank" pointer remains at the same place in the kill ring, so repeating `C-y` will yank another copy of the same old kill.

If you know how many `M-y` commands it would take to find the text you want, you can yank that text in one step using `C-y` with a numeric argument. `C-y` with an argument greater than one restores the text the specified number of entries back in the kill ring. Thus, `C-u 2 C-y` gets the next to the last block of killed text. It is equivalent to `C-y M-y`. `C-y` with a numeric argument starts counting from the "last yank" pointer, and sets the "last yank" pointer to the entry that it yanks.

The length of the kill ring is controlled by the variable `kill-ring-max`; no more than that many blocks of killed text are saved.

10.3 Accumulating Text

Usually we copy or move text by killing it and yanking it, but there are other ways that are useful for copying one block of text in many places, or for copying many scattered blocks of text into one place.

You can accumulate blocks of text from scattered locations either into a buffer or into a file if you like. These commands are described here. You can also use Emacs registers for storing and accumulating text. See Chapter 11 [Registers], page 63.

`C-x a` Append region to contents of specified buffer (`append-to-buffer`).

`M-x prepend-to-buffer`
: Prepend region to contents of specified buffer.

`M-x copy-to-buffer`
: Copy region into specified buffer, deleting that buffer's old contents.

`M-x insert-buffer`
: Insert contents of specified buffer into current buffer at point.

`M-x append-to-file`
: Append region to contents of specified file, at the end.

To accumulate text into a buffer, use the command `C-x a` *buffername* (`append-to-buffer`), which inserts a copy of the region into the buffer *buffername*, at the location of point in that buffer. If there is no buffer with that name, one is created. If you append text into a buffer which has been used for editing, the copied text goes into the middle of the text of the buffer, wherever point happens to be in it.

Point in that buffer is left at the end of the copied text, so successive uses of `C-x a` accumulate the text in the specified buffer in the same order as they were copied. Strictly speaking, `C-x a` does not always append to the text already in the buffer; but if `C-x a` is the only command used to alter a buffer, it does always append to the existing text because point is always at the end.

`M-x prepend-to-buffer` is just like `C-x a` except that point in the other buffer is left before the copied text, so successive prependings add text in reverse order. `M-x copy-to-buffer` is similar except that any existing text in the other buffer is deleted, so the buffer is left containing just the text newly copied into it.

You can retrieve the accumulated text from that buffer with `M-x insert-buffer`; this too takes *buffername* as an argument. It inserts a copy of the text in buffer *buffername* into the selected buffer. You could alternatively select the other buffer for editing, perhaps moving text from it by killing or with `C-x a`. See Chapter 16 [Buffers], page 105, for background information on buffers.

Instead of accumulating text within Emacs, in a buffer, you can append text directly into a file with `M-x append-to-file`, which takes *file-name* as an argument. It adds the text of the region to the end of the specified file. The file is changed immediately on disk. This command is normally used with files that are *not* being visited in Emacs. Using it on a file that Emacs is visiting can produce confusing results, because the text inside Emacs for that file will not change while the file itself changes.

10.4 Rectangles

The rectangle commands affect rectangular areas of the text: all the characters between a certain pair of columns, in a certain range of lines. Commands are provided to kill rectangles, yank killed rectangles, clear them out, or delete them. Rectangle commands are useful with text in multicolumnar formats, such as perhaps code with comments at the right, or for changing text into or out of such formats.

When you must specify a rectangle for a command to work on, you do it by putting the mark at one corner and point at the opposite corner. The rectangle thus specified is called the *region-rectangle* because it is controlled about the same way the region is controlled. But remember that a given combination of point and mark values can be interpreted either as specifying a region or as specifying a rectangle; it is up to the command that uses them to choose the interpretation.

`M-x delete-rectangle`
: Delete the text of the region-rectangle, moving any following text on each line leftward to the left edge of the region-rectangle.

`M-x kill-rectangle`
: Similar, but also save the contents of the region-rectangle as the "last killed rectangle".

`M-x yank-rectangle`
: Yank the last killed rectangle with its upper left corner at point.

`M-x open-rectangle`
: Insert blank space to fill the space of the region-rectangle. The previous contents of the region-rectangle are pushed rightward.

`M-x clear-rectangle`
: Clear the region-rectangle by replacing its contents with spaces.

The rectangle operations fall into two classes: commands deleting and moving rectangles, and commands for blank rectangles.

There are two ways to get rid of the text in a rectangle: you can discard the text (delete it) or save it as the "last killed" rectangle. The commands for these two ways are `M-x delete-rectangle` and `M-x kill-rectangle`. In either case, the portion of each line that falls inside the rectangle's boundaries is deleted, causing following text (if any) on the line to move left.

Note that "killing" a rectangle is not killing in the usual sense; the rectangle is not stored in the kill ring, but in a special place that can only record the most recent rectangle killed. This is because yanking a rectangle is so different from yanking linear text that different yank commands have to be used and yank-popping is hard to make sense of.

Inserting a rectangle is the opposite of deleting one. All you need to specify is where to put the upper left corner; that is done by putting point there. The rectangle's first line is inserted there, the rectangle's second line is inserted at a point one line vertically down, and so on. The number of lines affected is determined by the height of the saved rectangle.

To insert the last killed rectangle, type `M-x yank-rectangle`. This can be used to convert single-column lists into double-column lists; kill the second half of the list as a rectangle and then yank it beside the first line of the list.

There are two commands for working with blank rectangles: `M-x clear-rectangle` to blank out existing text, and `M-x open-rectangle` to insert a blank rectangle. Clearing a rectangle is equivalent to deleting it and then inserting as blank rectangle of the same size.

Rectangles can also be copied into and out of registers. See Section 11.3 [Rectangle Registers], page 64.

11 Registers

Emacs *registers* are places you can save text or positions for later use. Text saved in a register can be copied into the buffer once or many times; a position saved in a register is used by moving point to that position. Rectangles can also be copied into and out of registers (see Section 10.4 [Rectangles], page 59).

Each register has a name, which is a single character. A register can store either a piece of text or a position or a rectangle, but only one thing at any given time. Whatever you store in a register remains there until you store something else in that register.

`M-x view-register RET` *r*
: Display a description of what register *r* contains.

`M-x view-register` reads a register name as an argument and then displays the contents of the specified register.

11.1 Saving Positions in Registers

Saving a position records a spot in a buffer so that you can move back there later. Moving to a saved position reselects the buffer and moves point to the spot.

`C-x /` *r* Save location of point in register *r* (`point-to-register`).

`C-x j` *r* Jump to the location saved in register *r* (`register-to-point`).

To save the current location of point in a register, choose a name *r* and type `C-x /` *r*. The register *r* retains the location thus saved until you store something else in that register.

The command `C-x j` *r* moves point to the location recorded in register *r*. The register is not affected; it continues to record the same location. You can jump to the same position using the same register any number of times.

11.2 Saving Text in Registers

When you want to insert a copy of the same piece of text frequently, it may be impractical to use the kill ring, since each subsequent kill moves the piece of text further down on the ring. It becomes hard to keep track of what argument is needed to retrieve the same text with `C-y`. An alternative is to store the text in a register with `C-x x` (`copy-to-register`) and then retrieve it with `C-x g` (`insert-register`).

`C-x x` *r* Copy region into register *r* (`copy-to-register`).

`C-x g` *r* Insert text contents of register *r* (`insert-register`).

`C-x x` *r* stores a copy of the text of the region into the register named *r*. Given a numeric argument, `C-x x` deletes the text from the buffer as well.

`C-x g` *r* inserts in the buffer the text from register *r*. Normally it leaves point before the text and places the mark after, but with a numeric argument it puts point after the text and the mark before.

11.3 Saving Rectangles in Registers

A register can contain a rectangle instead of linear text. The rectangle is represented as a list of strings. See Section 10.4 [Rectangles], page 59, for basic information on rectangles and how rectangles in the buffer are specified.

`C-x r` *r* Copy the region-rectangle into register *r* (`copy-region-to-rectangle`). With numeric argument, delete it as well.

`C-x g` *r* Insert the rectangle stored in register *r* (if it contains a rectangle) (`insert-register`).

The `C-x g` command inserts linear text if the register contains that, or inserts a rectangle if the register contains one.

12 Controlling the Display

Since only part of a large buffer fits in the window, Emacs tries to show the part that is likely to be interesting. The display control commands allow you to specify which part of the text you want to see.

`C-l` Clear screen and redisplay, scrolling the selected window to center point vertically within it (`recenter`).

`C-v` Scroll forward (a windowful or a specified number of lines) (`scroll-up`).

`M-v` Scroll backward (`scroll-down`).

arg `C-l` Scroll so point is on line *arg* (`recenter`).

`C-x <` Scroll text in current window to the left (`scroll-left`).

`C-x >` Scroll to the right (`scroll-right`).

`C-x $` Make deeply indented lines invisible (`set-selective-display`).

12.1 Scrolling

If a buffer contains text that is too large to fit entirely within a window that is displaying the buffer, Emacs shows a contiguous section of the text. The section shown always contains point.

Scrolling means moving text up or down in the window so that different parts of the text are visible. Scrolling forward means that text moves up, and new text appears at the bottom. Scrolling backward moves text down and new text appears at the top.

Scrolling happens automatically if you move point past the bottom or top of the window. You can also explicitly request scrolling with the commands in this section.

The most basic scrolling command is `C-l` (`recenter`) with no argument. It clears the entire screen and redisplays all windows. In addition, the selected window is scrolled so that point is halfway down from the top of the window.

The scrolling commands `C-v` and `M-v` let you move all the text in the window up or down a few lines. `C-v` (`scroll-up`) with an argument shows you that many more lines at the bottom of the window, moving the text and point up together as `C-l` might. `C-v` with a negative argument shows you more lines at the top of the window. `Meta-v` (`scroll-down`) is like `C-v`, but moves in the opposite direction.

To read the buffer a windowful at a time, use `C-v` with no argument. It takes the last two lines at the bottom of the window and puts them at the top, followed by nearly a whole windowful of lines not previously visible. If point was in the text scrolled off the top, it moves to the new top of the window. `M-v` with no argument moves backward with overlap similarly. The number of lines of overlap across a `C-v` or `M-v` is controlled by the variable `next-screen-context-lines`; by default, it is two.

Another way to do scrolling is with `C-l` with a numeric argument. `C-l` does not clear the screen when given an argument; it only scrolls the selected window. With a positive argument *n*, it repositions text to put point *n* lines down from the top. An argument of zero puts point on the very top line. Point does not move with respect to the text; rather, the text and point move rigidly on the screen. `C-l` with a negative argument puts point that many lines from the bottom of the window. For example, `C-u - 1 C-l` puts point on the bottom line, and `C-u - 5 C-l` puts it five lines from the bottom. Just `C-u` as argument, as in `C-u C-l`, scrolls point to the center of the screen.

Scrolling happens automatically if point has moved out of the visible portion of the text when it is time to display. Usually the scrolling is done so as to put point vertically centered within the window. However, if the variable `scroll-step` has a nonzero value, an attempt is made to scroll the buffer by that many lines; if that is enough to bring point back into visibility, that is what is done.

12.2 Horizontal Scrolling

The text in a window can also be scrolled horizontally. This means that each line of text is shifted sideways in the window, and one or more characters at the beginning of each line are not displayed at all. When a window has been scrolled horizontally in this way, text lines are truncated rather than continued (see Section 4.7 [Continuation Lines], page 29), with a '`$`' appearing in the first column when there is text truncated to the left, and in the last column when there is text truncated to the right.

The command `C-x <` (`scroll-left`) scrolls the selected window to the left by *n* columns with argument *n*. With no argument, it scrolls by almost the full width of the window (two columns less, to be precise). `C-x >` (`scroll-right`) scrolls similarly to the right. The window cannot be scrolled any farther to the right once it is displaying normally (with each line starting at the window's left margin); attempting to do so has no effect.

12.3 Selective Display

Emacs has the ability to hide lines indented more than a certain number of columns (you specify how many columns). You can use this to get an overview of a part of a program.

To hide lines, type `C-x $` (`set-selective-display`) with a numeric argument *n*. (See Section 4.9 [Arguments], page 31, for how to give the argument.) Then lines with at least *n* columns of indentation disappear from the screen. The only indication of their presence is that three dots ('...') appear at the end of each visible line that is followed by one or more invisible ones.

The invisible lines are still present in the buffer, and most editing commands see them as usual, so it is very easy to put point in the middle of invisible text. When this happens, the cursor appears at the end of the previous line, after the three dots. If point is at the end of the visible line, before the newline that ends it, the cursor appears before the three dots.

The commands `C-n` and `C-p` move across the invisible lines as if they were not there.

To make everything visible again, type `C-x $` with no argument.

12.4 Variables Controlling Display

This section contains information for customization only. Beginning users should skip it.

The variable `mode-line-inverse-video` controls whether the mode line is displayed in inverse video (assuming the terminal supports it); `nil` means don't do so. See Chapter 2 [Mode Line], page 17.

If the variable `inverse-video` is non-`nil`, Emacs attempts to invert all the lines of the display from what they normally are.

If the variable `visible-bell` is non-`nil`, Emacs attempts to make the whole screen blink when it would normally make an audible bell sound. This variable has no effect if your terminal does not have a way to make the screen blink.

When you reenter Emacs after suspending, Emacs normally clears the screen and redraws the entire display. On some terminals with more than one page of memory, it is possible to arrange the termcap entry so that the '`ti`' and '`te`' strings (output to the terminal when Emacs is entered and exited, respectively) switch between pages of memory so as to use one page for Emacs and another page for other output. Then you might want to set the variable `no-redraw-on-reenter` non-`nil` so that Emacs will assume, when resumed, that the screen page it is using still contains what Emacs last wrote there.

The variable `echo-keystrokes` controls the echoing of multi-character keys; its value is the number of seconds of pause required to cause echoing to start, or zero meaning don't echo at all. See Section 1.2 [Echo Area], page 14.

If the variable `ctl-arrow` is `nil`, control characters in the buffer are displayed with octal escape sequences, all except newline and tab. Altering the value of `ctl-arrow` makes it local to the current buffer; until that time, the default value is in effect. The default is initially `t`. See Section 28.2.3 [Locals], page 220.

Normally, a tab character in the buffer is displayed as whitespace which extends to the next display tab stop position, and display tab stops come at intervals equal to eight spaces. The number of spaces per tab is controlled by the variable `tab-width`, which is made local by changing it, just like `ctl-arrow`. Note that how the tab character in the buffer is displayed has nothing to do with the definition of `TAB` as a command.

If you set the variable `selective-display-ellipses` to `nil`, the three dots do not appear at the end of a line that precedes invisible lines. Then there is no visible indication of the invisible lines. This variable too becomes local automatically when set.

13 Searching and Replacement

Like other editors, Emacs has commands for searching for occurrences of a string. The principal search command is unusual in that it is *incremental*; it begins to search before you have finished typing the search string. There are also nonincremental search commands more like those of other editors.

Besides the usual `replace-string` command that finds all occurrences of one string and replaces them with another, Emacs has a fancy replacement command called `query-replace` which asks interactively which occurrences to replace.

13.1 Incremental Search

An incremental search begins searching as soon as you type the first character of the search string. As you type in the search string, Emacs shows you where the string (as you have typed it so far) would be found. When you have typed enough characters to identify the place you want, you can stop. Depending on what you will do next, you may or may not need to terminate the search explicitly with an `ESC` first.

`C-s` Incremental search forward (`isearch-forward`).

`C-r` Incremental search backward (`isearch-backward`).

`C-s` starts an incremental search. `C-s` reads characters from the keyboard and positions the cursor at the first occurrence of the characters that you have typed. If you type `C-s` and then `F`, the cursor moves right after the first '`F`'. Type an `O`, and see the cursor move to after the first '`FO`'. After another `O`, the cursor is after the first '`FOO`' after the place where you started the search. Meanwhile, the search string '`FOO`' has been echoed in the echo area.

The echo area display ends with three dots when actual searching is going on. When search is waiting for more input, the three dots are removed. (On slow terminals, the three dots are not displayed.)

If you make a mistake in typing the search string, you can erase characters with `DEL`. Each `DEL` cancels the last character of search string. This does not happen until Emacs is ready to read another input character; first it must either find, or fail to find, the character you want to erase. If you do not want to wait for this to happen, use `C-g` as described below.

When you are satisfied with the place you have reached, you can type ESC, which stops searching, leaving the cursor where the search brought it. Also, any command not specially meaningful in searches stops the searching and is then executed. Thus, typing C-a would exit the search and then move to the beginning of the line. ESC is necessary only if the next command you want to type is a printing character, DEL, ESC, or another control character that is special within searches (C-q, C-w, C-r, C-s or C-y).

Sometimes you search for 'FOO' and find it, but not the one you expected to find. There was a second 'FOO' that you forgot about, before the one you were looking for. In this event, type another C-s to move to the next occurrence of the search string. This can be done any number of times. If you overshoot, you can cancel some C-s characters with DEL.

After you exit a search, you can search for the same string again by typing just C-s C-s: the first C-s is the key that invokes incremental search, and the second C-s means "search again".

If your string is not found at all, the echo area says 'Failing I-Search'. The cursor is after the place where Emacs found as much of your string as it could. Thus, if you search for 'FOOT', and there is no 'FOOT', you might see the cursor after the 'FOO' in 'FOOL'. At this point there are several things you can do. If your string was mistyped, you can rub some of it out and correct it. If you like the place you have found, you can type ESC or some other Emacs command to "accept what the search offered". Or you can type C-g, which removes from the search string the characters that could not be found (the 'T' in 'FOOT'), leaving those that were found (the 'FOO' in 'FOOT'). A second C-g at that point cancels the search entirely, returning point to where it was when the search started.

If a search is failing and you ask to repeat it by typing another C-s, it starts again from the beginning of the buffer. Repeating a failing reverse search with C-r starts again from the end. This is called *wrapping around*. 'Wrapped' appears in the search prompt once this has happened.

The C-g "quit" character does special things during searches; just what it does depends on the status of the search. If the search has found what you specified and is waiting for input, C-g cancels the entire search. The cursor moves back to where you started the search. If C-g is typed when there are characters in the search string that have not been found—because Emacs is still searching for them, or because it has failed to find them—then the search string characters which have not been found are discarded from the search string. With them gone, the search is now successful and waiting for more input, so a second C-g will cancel the entire search.

To search for a control character such as C-s or DEL or ESC, you must quote it by typing C-q first. This function of C-q is analogous to its meaning as an Emacs command: it causes the following character to be treated the way a graphic character would normally be treated in the same context.

You can change to searching backwards with `C-r`. If a search fails because the place you started was too late in the file, you should do this. Repeated `C-r` keeps looking for more occurrences backwards. A `C-s` starts going forwards again. `C-r` in a search can be cancelled with `DEL`.

If you know initially that you want to search backwards, you can use `C-r` instead of `C-s` to start the search, because `C-r` is also a key running a command (`isearch-backward`) to search backward.

The characters `C-y` and `C-w` can be used in incremental search to grab text from the buffer into the search string. This makes it convenient to search for another occurrence of text at point. `C-w` copies the word after point as part of the search string, advancing point over that word. Another `C-s` to repeat the search will then search for a string including that word. `C-y` is similar to `C-w` but copies all the rest of the current line into the search string.

All the characters special in incremental search can be changed by setting the following variables:

`search-delete-char`
: Character to delete from incremental search string (normally `DEL`).

`search-exit-char`
: Character to exit incremental search (normally `ESC`).

`search-quote-char`
: Character to quote special characters for incremental search (normally `C-q`).

`search-repeat-char`
: Character to repeat incremental search forwards (normally `C-s`).

`search-reverse-char`
: Character to repeat incremental search backwards (normally `C-r`).

`search-yank-line-char`
: Character to pull rest of line from buffer into search string (normally `C-y`).

`search-yank-word-char`
: Character to pull next word from buffer into search string (normally `C-w`).

13.1.1 Slow Terminal Incremental Search

Incremental search on a slow terminal uses a modified style of display that is designed to take less time. Instead of redisplaying the buffer at each place the search gets to, it creates a new single-line window and uses that to display the line that the search has found. The single-line window comes into play as soon as point gets outside of the text that is already on the screen.

When the search is terminated, the single-line window is removed. Only at this time is the window in which the search was done redisplayed to show its new value of point.

The three dots at the end of the search string, normally used to indicate that searching is going on, are not displayed in slow style display.

The slow terminal style of display is used when the terminal baud rate is less than or equal to the value of the variable `search-slow-speed`, initially 1200.

The number of lines to use in slow terminal search display is controlled by the variable `search-slow-window-lines`. 1 is its normal value.

13.2 Nonincremental Search

Emacs also has conventional nonincremental search commands, which require you to type the entire search string before searching begins.

`C-s ESC` *string* `RET`
: Search for *string*.

`C-r ESC` *string* `RET`
: Search backward for *string*.

To do a nonincremental search, first type `C-s ESC`. This enters the minibuffer to read the search string; terminate the string with `RET`, and then the search is done. If the string is not found the search command gets an error.

The way `C-s ESC` works is that the `C-s` invokes incremental search, which is specially programmed to invoke nonincremental search if the argument you give it is empty. (Such an empty argument would otherwise be useless.) `C-r ESC` also works this way.

Forward and backward nonincremental searches are implemented by the commands `search-forward` and `search-backward`. These commands may be bound to keys in the usual manner. The reason that incremental search is programmed to invoke them as well is that `C-s ESC` is the traditional sequence of characters used in Emacs to invoke nonincremental search.

However, nonincremental searches performed using `C-s ESC` do not call `search-forward` right away. The first thing done is to see if the next character is `C-w`, which requests a word search.

13.3 Word Search

Word search searches for a sequence of words without regard to how the words are separated. More precisely, you type a string of many words, using single spaces to separate them, and the string can be found even if there are multiple spaces, newlines or other punctuation between the words.

Word search is useful in editing documents formatted by text formatters. If you edit while looking at the printed, formatted version, you can't tell where the line breaks are in the source file. With word search, you can search without having to know them.

`C-s ESC C-w` *words* `RET`
: Search for *words*, ignoring differences in punctuation.

`C-r ESC C-w` *words* `RET`
: Search backward for *words*, ignoring differences in punctuation.

Word search is a special case of nonincremental search and is invoked with `C-s ESC C-w`. This is followed by the search string, which must always be terminated with `RET`. Being nonincremental, this search does not start until the argument is terminated. It works by constructing a regular expression and searching for that. See Section 13.4 [Regexp Search], page 73.

A backward word search can be done by `C-r ESC C-w`.

Forward and backward word searches are implemented by the commands `word-search-forward` and `word-search-backward`. These commands may be bound to keys in the usual manner. The reason that incremental search is programmed to invoke them as well is that `C-s ESC C-w` is the traditional Emacs sequence of keys for word search.

13.4 Regular Expression Search

A *regular expression* (*regexp*, for short) is a pattern that denotes a set of strings, possibly an infinite set. Searching for matches for a regexp is a very powerful operation that editors on Unix systems have traditionally offered. In GNU Emacs, you can search for the next match for a regexp either incrementally or not.

Incremental search for a regexp is done by typing `C-M-s` (`isearch-forward-regexp`). This command reads a search string incrementally just like `C-s`, but it treats the search string as a regexp rather than looking for an exact match against the text in the buffer. Each time you add text to the search string, you make the regexp longer, and the new regexp is searched for. A reverse regexp search command `isearch-backward-regexp` also exists but no key runs it.

All of the control characters that do special things within an ordinary incremental search have the same function in incremental regexp search. Typing `C-s` or `C-r` immediately after starting the search retrieves the last incremental search regexp used; that is to say, incremental regexp and non-regexp searches have independent defaults.

Note that adding characters to the regexp in an incremental regexp search does not make the cursor move back and start again. Perhaps it ought to; I am not sure. As it stands, if you have searched for '`foo`' and you add '`\|bar`', the search will not check for a '`bar`' in the buffer before the '`foo`'.

Nonincremental search for a regexp is done by the functions `re-search-forward` and `re-search-backward`. You can invoke these with `M-x`, or bind them to keys. Also, you can call `re-search-forward` by way of incremental regexp search with `C-M-s ESC`.

13.5 Syntax of Regular Expressions

Regular expressions have a syntax in which a few characters are special constructs and the rest are *ordinary*. An ordinary character is a simple regular expression which matches that character and nothing else. The special characters are '`$`', '`^`', '`.`', '`*`', '`+`', '`?`', '`[`', '`]`' and '`\`'; no new special characters will be defined. Any other character appearing in a regular expression is ordinary, unless a '`\`' precedes it.

For example, '`f`' is not a special character, so it is ordinary, and therefore '`f`' is a regular expression that matches the string '`f`' and no other string. (It does *not* match the string '`ff`'.) Likewise, '`o`' is a regular expression that matches only '`o`'.

Any two regular expressions *a* and *b* can be concatenated. The result is a regular expression which matches a string if *a* matches some amount of the beginning of that string and *b* matches the rest of the string.

As a simple example, we can concatenate the regular expressions 'f' and 'o' to get the regular expression 'fo', which matches only the string 'fo'. Still trivial. To do something nontrivial, you need to use one of the special characters. Here is a list of them.

. (Period)
: is a special character that matches any single character except a newline. Using concatenation, we can make regular expressions like 'a.b' which matches any three-character string which begins with 'a' and ends with 'b'.

*
: is not a construct by itself; it is a suffix, which means the preceding regular expression is to be repeated as many times as possible. In 'fo*', the '*' applies to the 'o', so 'fo*' matches one 'f' followed by any number of 'o's. The case of zero 'o's is allowed: 'fo*' does match 'f'.

  '*' always applies to the *smallest* possible preceding expression. Thus, 'fo*' has a repeating 'o', not a repeating 'fo'.

  The matcher processes a '*' construct by matching, immediately, as many repetitions as can be found. Then it continues with the rest of the pattern. If that fails, backtracking occurs, discarding some of the matches of the '*'-modified construct in case that makes it possible to match the rest of the pattern. For example, matching 'ca*ar' against the string 'caaar', the 'a*' first tries to match all three 'a's; but the rest of the pattern is 'ar' and there is only 'r' left to match, so this try fails. The next alternative is for 'a*' to match only two 'a's. With this choice, the rest of the regexp matches successfully.

+
: Is a suffix character similar to '*' except that it requires that the preceding expression be matched at least once. So, for example, 'ca+r' will match the strings 'car' and 'caaaar' but not the string 'cr', whereas 'ca*r' would match all three strings.

?
: Is a suffix character similar to '*' except that it can match the preceding expression either once or not at all. For example, 'ca?r' will match 'car' or 'cr'; nothing else.

[...]
: '[' begins a *character set*, which is terminated by a ']'. In the simplest case, the characters between the two form the set. Thus, '[ad]' matches either one 'a' or one 'd', and '[ad]*' matches any string composed of just 'a's and 'd's (including the empty string), from which it follows that 'c[ad]*r' matches 'cr', 'car', 'cdr', 'caddaar', etc.

  Character ranges can also be included in a character set, by writing two characters with a '-' between them. Thus, '[a-z]' matches any lower-case letter. Ranges may be intermixed freely with individual characters, as in '[a-z$%.]', which matches any lower case letter or '$', '%' or period.

  Note that the usual special characters are not special any more inside a character set. A completely different set of special characters exists inside character sets: ']', '-' and '^'.

  To include a ']' in a character set, you must make it the first character. For example, '[]a]' matches ']' or 'a'. To include a '-', write '---', which is a range containing only '-'. To include '^', make it other than the first character in the set.

`[^ ... ]` '`[^`' begins a *complement character set*, which matches any character except the ones specified. Thus, '`[^a-z0-9A-Z]`' matches all characters *except* letters and digits.

'`^`' is not special in a character set unless it is the first character. The character following the '`^`' is treated as if it were first ('`-`' and '`]`' are not special there).

Note that a complement character set can match a newline, unless newline is mentioned as one of the characters not to match.

`^` is a special character that matches the empty string, but only if at the beginning of a line in the text being matched. Otherwise it fails to match anything. Thus, '`^foo`' matches a '`foo`' which occurs at the beginning of a line.

`$` is similar to '`^`' but matches only at the end of a line. Thus, '`xx*$`' matches a string of one '`x`' or more at the end of a line.

`\` has two functions: it quotes the special characters (including '`\`'), and it introduces additional special constructs.

Because '`\`' quotes special characters, '`\$`' is a regular expression which matches only '`$`', and '`\[`' is a regular expression which matches only '`[`', and so on.

Note: for historical compatibility, special characters are treated as ordinary ones if they are in contexts where their special meanings make no sense. For example, '`*foo`' treats '`*`' as ordinary since there is no preceding expression on which the '`*`' can act. It is poor practice to depend on this behavior; better to quote the special character anyway, regardless of where is appears.

For the most part, '`\`' followed by any character matches only that character. However, there are several exceptions: characters which, when preceded by '`\`', are special constructs. Such characters are always ordinary when encountered on their own. Here is a table of '`\`' constructs.

`\|` specifies an alternative. Two regular expressions *a* and *b* with '`\|`' in between form an expression that matches anything that either *a* or *b* will match.

Thus, '`foo\|bar`' matches either '`foo`' or '`bar`' but no other string.

'`\|`' applies to the largest possible surrounding expressions. Only a surrounding '`\( ... \)`' grouping can limit the grouping power of '`\|`'.

Full backtracking capability exists to handle multiple uses of '`\|`'.

`\( ... \)` is a grouping construct that serves three purposes:

1. To enclose a set of '`\|`' alternatives for other operations. Thus, '`\(foo\|bar\)x`' matches either '`foox`' or '`barx`'.
2. To enclose a complicated expression for the postfix '`*`' to operate on. Thus, '`ba\(na\)*`' matches '`bananana`', etc., with any (zero or more) number of '`na`' strings.

3. To mark a matched substring for future reference.

This last application is not a consequence of the idea of a parenthetical grouping; it is a separate feature which happens to be assigned as a second meaning to the same ‘`\( … \)`’ construct because there is no conflict in practice between the two meanings. Here is an explanation of this feature:

`\`*digit*
: after the end of a ‘`\( … \)`’ construct, the matcher remembers the beginning and end of the text matched by that construct. Then, later on in the regular expression, you can use ‘`\`’ followed by *digit* to mean “match the same text matched the *digit*’th time by the ‘`\( … \)`’ construct.”

  The strings matching the first nine ‘`\( … \)`’ constructs appearing in a regular expression are assigned numbers 1 through 9 in order that the open-parentheses appear in the regular expression. ‘`\1`’ through ‘`\9`’ may be used to refer to the text matched by the corresponding ‘`\( … \)`’ construct.

  For example, ‘`\(.*\)\1`’ matches any newline-free string that is composed of two identical halves. The ‘`\(.*\)`’ matches the first half, which may be anything, but the ‘`\1`’ that follows must match the same exact text.

``\` ``
: matches the empty string, provided it is at the beginning of the buffer.

`\'`
: matches the empty string, provided it is at the end of the buffer.

`\b`
: matches the empty string, provided it is at the beginning or end of a word. Thus, ‘`\bfoo\b`’ matches any occurrence of ‘`foo`’ as a separate word. ‘`\bballs?\b`’ matches ‘`ball`’ or ‘`balls`’ as a separate word.

`\B`
: matches the empty string, provided it is *not* at the beginning or end of a word.

`\<`
: matches the empty string, provided it is at the beginning of a word.

`\>`
: matches the empty string, provided it is at the end of a word.

`\w`
: matches any word-constituent character. The editor syntax table determines which characters these are.

`\W`
: matches any character that is not a word-constituent.

`\s`*code*
: matches any character whose syntax is *code*. *code* is a character which represents a syntax code: thus, ‘`w`’ for word constituent, ‘`-`’ for whitespace, ‘`(`’ for open-parenthesis, etc. See Section 28.5 [Syntax], page 229.

`\S`*code*
: matches any character whose syntax is not *code*.

Here is a complicated regexp, used by Emacs to recognize the end of a sentence together with any whitespace that follows. It is given in Lisp syntax to enable you to distinguish the spaces from the tab characters. In Lisp syntax, the string constant begins and ends with a double-quote. ‘`\"`’ stands for a double-quote as part of the regexp, ‘`\\`’ for a backslash as part of the regexp, ‘`\t`’ for a tab and ‘`\n`’ for a newline.

```
"[.?!][]\"')]*\\($\\|\t\\|  \\)[ \t\n]*"
```

This contains four parts in succession: a character set matching period, '`?`' or '`!`'; a character set matching close-brackets, quotes or parentheses, repeated any number of times; an alternative in backslash-parentheses that matches end-of-line, a tab or two spaces; and a character set matching whitespace characters, repeated any number of times.

13.6 Searching and Case

All sorts of searches in Emacs normally ignore the case of the text they are searching through; if you specify searching for '`FOO`', then '`Foo`' and '`foo`' are also considered a match. Regexps, and in particular character sets, are included: '`[aB]`' would match '`a`' or '`A`' or '`b`' or '`B`'.

If you do not want this feature, set the variable `case-fold-search` to `nil`. Then all letters must match exactly, including case. This is a per-buffer variable; altering the variable affects only the current buffer, but there is a default value which you can change as well. See Section 28.2.3 [Locals], page 220.

13.7 Replacement Commands

Global search-and-replace operations are not needed as often in Emacs as they are in other editors, but they are available. In addition to the simple `replace-string` command which is like that found in most editors, there is a `query-replace` command which asks you, for each occurrence of the pattern, whether to replace it.

The replace commands all replace one string (or regexp) with one replacement string. It is possible to perform several replacements in parallel using the command `expand-region-abbrevs`. See Section 23.2 [Expanding Abbrevs], page 178.

13.7.1 Unconditional Replacement

`M-x replace-string RET` *string* `RET` *newstring* `RET`
: Replace every occurrence of *string* with *newstring*.

`M-x replace-regexp RET` *regexp* `RET` *newstring* `RET`
: Replace every match for *regexp* with *newstring*.

To replace every instance of '`foo`' after point with '`bar`', use the command `M-x replace-string` with the two arguments '`foo`' and '`bar`'. Replacement occurs only after point, so if you want to cover the whole buffer you must go to the beginning first. All occurrences up to the end of the buffer are replaced; to limit replacement to part of the buffer, narrow to that part of the buffer before doing the replacement (see Section 27.2 [Narrowing], page 208).

When `replace-string` exits, point is left at the last occurrence replaced. The value of point when the `replace-string` command was issued is remembered on the mark ring; `C-u C-SPC` moves back there.

A numeric argument restricts replacement to matches that are surrounded by word boundaries.

13.7.2 Regexp Replacement

`replace-string` replaces exact matches for a single string. The similar command `replace-regexp` replaces any match for a specified pattern.

In `replace-regexp`, the *newstring* need not be constant. It can refer to all or part of what is matched by the *regexp*. '`\&`' in *newstring* stands for the entire text being replaced. '`\`*d*' in *newstring*, where *d* is a digit, stands for whatever matched the *d*'th parenthesized grouping in *regexp*. For example,

```
M-x replace-regexp RET c[ad]+r RET \&-safe RET
```

would replace (for example) '`cadr`' with '`cadr-safe`' and '`cddr`' with '`cddr-safe`'.

```
M-x replace-regexp RET \(c[ad]+r\)-safe RET \1 RET
```

would perform exactly the opposite replacements. To include a '`\`' in the text to replace with, you must give '`\\`'.

13.7.3 Replace Commands and Case

If the arguments to a replace command are in lower case, it preserves case when it makes a replacement. Thus, the command

```
M-x replace-string RET foo RET bar RET
```

replaces a lower case '`foo`' with a lower case '`bar`', '`FOO`' with '`BAR`', and '`Foo`' with '`Bar`'. If upper case letters are used in the second argument, they remain upper case every time that argument is inserted. If upper case letters are used in the first argument, the second argument is always substituted exactly as given, with no case conversion. Likewise, if the variable `case-replace` is set to `nil`, replacement is done without case conversion. If `case-fold-search` is set to `nil`, case is significant in matching occurrences of '`foo`' to replace; also, case conversion of the replacement string is not done.

13.7.4 Query Replace

`M-%` *string* `RET` *newstring* `RET`
`M-x query-replace RET` *string* `RET` *newstring* `RET`
: Replace some occurrences of *string* with *newstring*.

`M-x query-replace-regexp RET` *regexp* `RET` *newstring* `RET`
: Replace some matches for *regexp* with *newstring*.

If you want to change only some of the occurrences of '`foo`' to '`bar`', not all of them, then you cannot use an ordinary `replace-string`. Instead, use `M-%` (`query-replace`). This command finds occurrences of '`foo`' one by one, displays each occurrence and asks you whether to replace it. A numeric argument to `query-replace` tells it to consider only occurrences that are bounded by word-delimiter characters.

Aside from querying, `query-replace` works just like `replace-string`, and `query-replace-regexp` works just like `replace-regexp`.

The things you can type when you are shown an occurrence of *string* or a match for *regexp* are:

`SPC`
: to replace the occurrence with *newstring*. This preserves case, just like `replace-string`, provided `case-replace` is non-`nil`, as it normally is.

`DEL`
: to skip to the next occurrence without replacing this one.

`,` (Comma)
: to replace this occurrence and display the result. You are then asked for another input character, except that since the replacement has already been made, `DEL` and `SPC` are equivalent. You could type `C-r` at this point (see below) to alter the replaced text. You could also type `C-x u` to undo the replacement; this exits the `query-replace`, so

if you want to do further replacement you must use `C-x ESC` to restart (see Section 6.4 [Repetition], page 39).

`ESC` to exit without doing any more replacements.

`.` (Period) to replace this occurrence and then exit.

`!` to replace all remaining occurrences without asking again.

`^` to go back to the location of the previous occurrence (or what used to be an occurrence), in case you changed it by mistake. This works by popping the mark ring. Only one `^` in a row is allowed, because only one previous replacement location is kept during `query-replace`.

`C-r` to enter a recursive editing level, in case the occurrence needs to be edited rather than just replaced with *newstring*. When you are done, exit the recursive editing level with `C-M-c` and the next occurrence will be displayed. See Section 27.1 [Recursive Edit], page 207.

`C-w` to delete the occurrence, and then enter a recursive editing level as in `C-r`. Use the recursive edit to insert text to replace the deleted occurrence of *string*. When done, exit the recursive editing level with `C-M-c` and the next occurrence will be displayed.

`C-l` to redisplay the screen and then give another answer.

`C-h` to display a message summarizing these options, then give another answer.

If you type any other character, the `query-replace` is exited, and the character executed as a command. To restart the `query-replace`, use `C-x ESC`, which repeats the `query-replace` because it used the minibuffer to read its arguments. See Section 6.4 [Repetition], page 39.

13.8 Other Search-and-Loop Commands

Here are some other commands that find matches for a regular expression. They all operate from point to the end of the buffer.

`M-x occur` Print each line that follows point and contains a match for the specified regexp. A numeric argument specifies the number of context lines to print before and after each matching line; the default is none.

The buffer '`*Occur*`' containing the output serves as a menu for finding the occurrences in their original context. Find an occurrence as listed in '`*Occur*`', position point there and type `C-c C-c`; this switches to the buffer that was searched and moves point to the original of the same occurrence.

`M-x list-matching-lines`
Synonym for `M-x occur`.

`M-x count-matches`
: Print the number of matches following point for the specified regexp.

`M-x delete-non-matching-lines`
: Delete each line that follows point and does not contain a match for the specified regexp.

`M-x delete-matching-lines`
: Delete each line that follows point and contains a match for the specified regexp.

14 Commands for Fixing Typos

In this chapter we describe the commands that are especially useful for the times when you catch a mistake in your text just after you have made it, or change your mind while composing text on line.

14.1 Killing Your Mistakes

`DEL` Delete last character (`delete-backward-char`).

`M-DEL` Kill last word (`backward-kill-word`).

`C-x DEL` Kill to beginning of sentence (`backward-kill-sentence`).

The `DEL` character (`delete-backward-char`) is the most important correction command. When used among graphic (self-inserting) characters, it can be thought of as canceling the last character typed.

When your mistake is longer than a couple of characters, it might be more convenient to use `M-DEL` or `C-x DEL`. `M-DEL` kills back to the start of the last word, and `C-x DEL` kills back to the start of the last sentence. `C-x DEL` is particularly useful when you are thinking of what to write as you type it, in case you change your mind about phrasing. `M-DEL` and `C-x DEL` save the killed text for `C-y` and `M-y` to retrieve. See Section 10.2 [Yanking], page 55.

`M-DEL` is often useful even when you have typed only a few characters wrong, if you know you are confused in your typing and aren't sure exactly what you typed. At such a time, you cannot correct with `DEL` except by looking at the screen to see what you did. It requires less thought to kill the whole word and start over again.

14.2 Transposing Text

`C-t` Transpose two characters (`transpose-chars`).

`M-t` Transpose two words (`transpose-words`).

`C-M-t` Transpose two balanced expressions (`transpose-sexps`).

`C-x C-t` Transpose two lines (`transpose-lines`).

The common error of transposing two characters can be fixed, when they are adjacent, with the `C-t` command (`transpose-chars`). Normally, `C-t` transposes the two characters on either side of point. When given at the end of a line, rather than transposing the last character of the line with the newline, which would be useless, `C-t` transposes the last two characters on the line. So, if you catch your transposition error right away, you can fix it with just a `C-t`. If you don't catch it so fast, you must move the cursor back to between the two transposed characters. If you transposed a space with the last character of the word before it, the word motion commands are a good way of getting there. Otherwise, a reverse search (`C-r`) is often the best way. See Chapter 13 [Search], page 69.

`Meta-t` (`transpose-words`) transposes the word before point with the word after point. It moves point forward over a word, dragging the word preceding or containing point forward as well. The punctuation characters between the words do not move. For example, '`FOO, BAR`' transposes into '`BAR, FOO`' rather than '`BAR FOO,`'.

`C-M-t` (`transpose-sexps`) is a similar command for transposing two expressions (see Section 21.2 [Lists], page 140), and `C-x C-t` (`transpose-lines`) exchanges lines. They work like `M-t` except in determining the division of the text into syntactic units.

A numeric argument to a transpose command serves as a repeat count: it tells the transpose command to move the character (word, sexp, line) before or containing point across several other characters (words, sexps, lines). For example, `C-u 3 C-t` moves the character before point forward across three other characters. This is equivalent to repeating `C-t` three times. `C-u - 4 M-t` moves the word before point backward across four words. `C-u - C-M-t` would cancel the effect of plain `C-M-t`.

A numeric argument of zero is assigned a special meaning (because otherwise a command with a repeat count of zero would do nothing): to transpose the character (word, sexp, line) ending after point with the one ending after the mark.

14.3 Case Conversion

`M-- M-l` Convert last word to lower case. Note `Meta--` is Meta-minus.

`M-- M-u` Convert last word to all upper case.

`M-- M-c` Convert last word to lower case with capital initial.

A very common error is to type words in the wrong case. Because of this, the word case-conversion commands `M-l`, `M-u` and `M-c` have a special feature when used with a negative argument:

they do not move the cursor. As soon as you see you have mistyped the last word, you can simply case-convert it and go on typing. See Section 20.7 [Case], page 137.

14.4 Checking and Correcting Spelling

`M-$` Check and correct spelling of word (`spell-word`).

`M-x spell-buffer`
  Check and correct spelling of each word in the buffer.

`M-x spell-region`
  Check and correct spelling of each word in the region.

`M-x spell-string`
  Check spelling of specified word.

To check the spelling of the word before point, and optionally correct it as well, use the command `M-$` (`spell-word`). This command runs an inferior process containing the `spell` program to see whether the word is correct English. If it is not, it asks you to edit the word (in the minibuffer) into a corrected spelling, and then does a `query-replace` to substitute the corrected spelling for the old one throughout the buffer.

If you exit the minibuffer without altering the original spelling, it means you do not want to do anything to that word. Then the `query-replace` is not done.

`M-x spell-buffer` checks each word in the buffer the same way that `spell-word` does, doing a `query-replace` if appropriate for every incorrect word.

`M-x spell-region` is similar but operates only on the region, not the entire buffer.

`M-x spell-string` reads a string as an argument and checks whether that is a correctly spelled English word. It prints in the echo area a message giving the answer.

15 File Handling

The basic unit of stored data in Unix is the *file*. To edit a file, you must tell Emacs to examine the file and prepare a buffer containing a copy of the file's text. This is called *visiting* the file. Editing commands apply directly to text in the buffer; that is, to the copy inside Emacs. Your changes appear in the file itself only when you *save* the buffer back into the file.

In addition to visiting and saving files, Emacs can delete, copy, rename, and append to files, and operate on file directories.

15.1 File Names

Most Emacs commands that operate on a file require you to specify the file name. (Saving and reverting are exceptions; the buffer knows which file name to use for them.) File names are specified using the minibuffer (see Chapter 6 [Minibuffer], page 35). *Completion* is available, to make it easier to specify long file names. See Section 6.3.2 [Completion], page 38.

There is always a *default file name* which will be used if you type just `RET`, entering an empty argument. Normally the default file name is the name of the file visited in the current buffer; this makes it easy to operate on that file with any of the Emacs file commands.

Each buffer has a default directory, normally the same as the directory of the file visited in that buffer. When Emacs reads a file name, if you do not specify a directory, the default directory is used. If you specify a directory in a relative fashion, with a name that does not start with a slash, it is interpreted with respect to the default directory. The default directory is kept in the variable `default-directory`, which has a separate value in every buffer.

For example, if the default file name is '`/u/rms/gnu/gnu.tasks`' then the default directory is '`/u/rms/gnu/`'. If you type just '`foo`', which does not specify a directory, it is short for '`/u/rms/gnu/foo`'. '`../.login`' would stand for '`/u/rms/.login`'. '`new/foo`' would stand for the filename '`/u/rms/gnu/new/foo`'.

The command `M-x pwd` prints the current buffer's default directory, and the command `M-x cd` sets it (to a value read using the minibuffer). A buffer's default directory changes only when the `cd` command is used. A file-visiting buffer's default directory is initialized to the directory of the file that is visited there. If a buffer is made randomly with `C-x b`, its default directory is copied from that of the buffer that was current at the time.

The default directory actually appears in the minibuffer when the minibuffer becomes active to read a file name. This serves two purposes: it shows you what the default is, so that you can type a relative file name and know with certainty what it will mean, and it allows you to edit the default to specify a different directory. This insertion of the default directory is inhibited if the variable `insert-default-directory` is set to `nil`.

Note that it is legitimate to type an absolute file name after you enter the minibuffer, ignoring the presence of the default directory name as part of the text. The final minibuffer contents may look invalid, but that is not so. See Section 6.1 [Minibuffer File], page 35.

'`$`' in a file name is used to substitute environment variables. For example, if you have used the shell command '`setenv FOO rms/hacks`' to set up an environment variable named '`FOO`', then you can use '`/u/$FOO/test.c`' or '`/u/${FOO}/test.c`' as an abbreviation for '`/u/rms/hacks/test.c`'. The environment variable name consists of all the alphanumeric characters after the '`$`'; alternatively, it may be enclosed in braces after the '`$`'. Note that the '`setenv`' command affects Emacs only if done before Emacs is started.

To access a file with '`$`' in its name, type '`$$`'. This pair is converted to a single '`$`' at the same time as variable substitution is performed for single '`$`'. The Lisp function that performs the substitution is called `substitute-in-file-name`. The substitution is performed only on filenames read as such using the minibuffer.

15.2 Visiting Files

`C-x C-f` Visit a file (`find-file`).

`C-x C-v` Visit a different file instead of the one visited last (`find-alternate-file`).

`C-x 4 C-f` Visit a file, in another window (`find-file-other-window`). Don't change this window.

Visiting a file means copying its contents into Emacs where you can edit them. Emacs makes a new buffer for each file that you visit. We say that the buffer is visiting the file that it was created to hold. Emacs constructs the buffer name from the file name by throwing away the directory, keeping just the name proper. For example, a file named '`/usr/rms/emacs.tex`' would get a buffer named '`emacs.tex`'. If there is already a buffer with that name, a unique name is constructed by appending '`<2>`', '`<3>`', or so on, using the lowest number that makes a name that is not already in use.

Each window's mode line shows the name of the buffer that is being displayed in that window, so you can always tell what buffer you are editing.

The changes you make with Emacs are made in the Emacs buffer. They do not take effect in the file that you visited, or any place permanent, until you *save* the buffer. Saving the buffer means that Emacs writes the current contents of the buffer into its visited file. See Section 15.3 [Saving], page 90.

If a buffer contains changes that have not been saved, the buffer is said to be *modified*. This is important because it implies that some changes will be lost if the buffer is not saved. The mode line displays two stars near the left margin if the buffer is modified.

To visit a file, use the command `C-x C-f` (`find-file`). Follow the command with the name of the file you wish to visit, terminated by a `RET`.

The file name is read using the minibuffer (see Chapter 6 [Minibuffer], page 35), with defaulting and completion in the standard manner (see Section 15.1 [File Names], page 87). While in the minibuffer, you can abort `C-x C-f` by typing `C-g`.

Your confirmation that `C-x C-f` has completed successfully is the appearance of new text on the screen and a new buffer name in the mode line. If the specified file does not exist and could not be created, or cannot be read, then an error results. The error message is printed in the echo area, and includes the file name which Emacs was trying to visit.

If you visit a file that is already in Emacs, `C-x C-f` does not make another copy. It selects the existing buffer containing that file. However, before doing so, it checks that the file itself has not changed since you visited or saved it last. If the file has changed, a warning message is printed. See Section 15.3.2 [Simultaneous Editing], page 94.

What if you want to create a file? Just visit it. Emacs prints '`(New File)`' in the echo area, but in other respects behaves as if you had visited an existing empty file. If you make any changes and save them, the file is created.

If you visit a nonexistent file unintentionally (because you typed the wrong file name), use the `C-x C-v` (`find-alternate-file`) command to visit the file you wanted. `C-x C-v` is similar to `C-x C-f`, but it kills the current buffer (after first offering to save it if it is modified). `C-x C-v` is allowed even if the current buffer is not visiting a file.

If the file you specify is actually a directory, Dired is called on that directory (see Section 15.7 [Dired], page 99). This can be inhibited by setting the variable `find-file-run-dired` to `nil`; then it is an error to try to visit a directory.

`C-x 4 f` (`find-file-other-window`) is like `C-x C-f` except that the buffer containing the specified file is selected in another window. The window that was selected before `C-x 4 f` continues to show the same buffer it was already showing. If this command is used when only one window is being displayed, that window is split in two, with one window showing the same before as before, and the other one showing the newly requested file. See Chapter 17 [Windows], page 111.

There are two hook variables that allow extensions to modify the operation of visiting files. Visiting a file that does not exist runs the functions in the list `find-file-not-found-hooks`; the value of this variable is expected to be a list of functions, and the functions are called one by one until one of them returns non-`nil`. Any visiting of a file, whether extant or not, expects `find-file-hooks` to contain list of functions and calls them all, one by one. In both cases the functions receive no arguments. Visiting a nonexistent file runs the `find-file-not-found-hooks` first.

15.3 Saving Files

Saving a buffer in Emacs means writing its contents back into the file that was visited in the buffer.

`C-x C-s`
: Save the current buffer in its visited file (`save-buffer`).

`C-x s`
: Save any or all buffers in their visited files (`save-some-buffers`).

`M-~`
: Forget that the current buffer has been changed (`not-modified`).

`C-x C-w`
: Save the current buffer in a specified file, and record that file as the one visited in the buffer (`write-file`).

`M-x set-visited-file-name`
: Change file the name under which the current buffer will be saved.

When you wish to save the file and make your changes permanent, type `C-x C-s` (`save-buffer`). After saving is finished, `C-x C-s` prints a message such as

```
Wrote /u/rms/gnu/gnu.tasks
```

If the selected buffer is not modified (no changes have been made in it since the buffer was created or last saved), saving is not really done, because it would have no effect. Instead, `C-x C-s` prints a message in the echo area saying

```
(No changes need to be written)
```

The command `C-x s` (`save-some-buffers`) can save any or all modified buffers. First it asks, for each modified buffer, whether to save it. These questions should be answered with `y` or `n`. `C-x C-c`, the key that kills Emacs, invokes `save-some-buffers` and therefore asks the same questions.

If you have changed a buffer and do not want the changes to be saved, you should take some action to prevent it. Otherwise, each time you use `save-some-buffers` you are liable to save it by mistake. One thing you can do is type `M-~` (`not-modified`), which clears out the indication that the buffer is modified. If you do this, none of the save commands will believe that the buffer needs to be saved. ('`~`' is often used as a mathematical symbol for 'not'; thus `Meta-~` is 'not', metafied.) You could also use `set-visited-file-name` (see below) to mark the buffer as visiting a different file name, one which is not in use for anything important. Alternatively, you can undo all the changes made since the file was visited or saved, by reading the text from the file again. This is called *reverting*. See Section 15.4 [Reverting], page 95. You could also undo all the changes by repeating the undo command `C-x u` until you have undone all the changes; but this only works if you have not made more changes than the undo mechanism can remember.

`M-x set-visited-file-name` alters the name of the file that the current buffer is visiting. It reads the new file name using the minibuffer. It can be used on a buffer that is not visiting a file, too. The buffer's name is changed to correspond to the file it is now visiting in the usual fashion (unless the new name is in use already for some other buffer; in that case, the buffer name is not changed). `set-visited-file-name` does not save the buffer in the newly visited file; it just alters the records inside Emacs so that, if you save the buffer, it will be saved in that file. It also marks the buffer as "modified" so that `C-x C-s` *will* save.

If you wish to mark the buffer as visiting a different file and save it right away, use `C-x C-w` (`write-file`). It is precisely equivalent to `set-visited-file-name` followed by `C-x C-s`. `C-x C-s` used on a buffer that is not visiting with a file has the same effect as `C-x C-w`; that is, it reads a file name, marks the buffer as visiting that file, and saves it there. The default file name in a buffer that is not visiting a file is made by combining the buffer name with the buffer's default directory.

If Emacs is about to save a file and sees that the date of the latest version on disk does not match what Emacs last read or wrote, Emacs notifies you of this fact, because it probably indicates a problem caused by simultaneous editing and requires your immediate attention. See Section 15.3.2 [Simultaneous Editing], page 94.

If the variable `require-final-newline` is non-`nil`, Emacs puts a newline at the end of any file that doesn't already end in one, every time a file is saved or written.

You can implement other ways to write files, and other things to be done before writing them, using the hook variable `write-file-hooks`. The value of this variable should be a list of Lisp functions. When a file is to be written, the functions in the list are called, one by one, with no arguments. If one of them returns a non-`nil` value, Emacs takes this to mean that the file has been written in some suitable fashion; the rest of the functions are not called, and normal writing is not done.

15.3.1 Backup Files

Because Unix does not provide version numbers in file names, rewriting a file in Unix automatically destroys all record of what the file used to contain. Thus, saving a file from Emacs throws away the old contents of the file—or it would, except that Emacs carefully copies the old contents to another file, called the *backup* file, before actually saving. (Provided the variable `make-backup-files` is non-`nil`. Backup files are not written if this variable is `nil`).

At your option, Emacs can keep either a single backup file or a series of numbered backup files for each file that you edit.

Emacs makes a backup for a file only the first time the file is saved from one buffer. No matter how many times you save a file, its backup file continues to contain the contents from before the file was visited. Normally this means that the backup file contains the contents from before the current editing session; however, if you kill the buffer and then visit the file again, a new backup file will be made by the next save.

15.3.1.1 Single or Numbered Backups

If you choose to have a single backup file (this is the default), the backup file's name is constructed by appending '`~`' to the file name being edited; thus, the backup file for '`eval.c`' would be '`eval.c~`'.

If you choose to have a series of numbered backup files, backup file names are made by appending '`.~`', the number, and another '`~`' to the original file name. Thus, the backup files of '`eval.c`' would be called '`eval.c.~1~`', '`eval.c.~2~`', and so on, through names like '`eval.c.~259~`' and beyond.

If protection stops you from writing backup files under the usual names, the backup file is written as '`%backup%~`' in your home directory. Only one such file can exist, so only the most recently made such backup is available.

The choice of single backup or numbered backups is controlled by the variable `version-control`. Its possible values are

`t` Make numbered backups.

`nil` Make numbered backups for files that have numbered backups already. Otherwise, make single backups.

`never` Do not in any case make numbered backups; always make single backups.

`version-control` may be set locally in an individual buffer to control the making of backups for that buffer's file. For example, Rmail mode locally sets `version-control` to `never` to make sure that there is only one backup for an Rmail file. See Section 28.2.3 [Locals], page 220.

15.3.1.2 Automatic Deletion of Backups

To prevent unlimited consumption of disk space, Emacs can delete numbered backup versions automatically. Generally Emacs keeps the first few backups and the latest few backups, deleting any in between. This happens every time a new backup is made. The two variables that control the deletion are `kept-old-versions` and `kept-new-versions`. Their values are, respectively the number of oldest (lowest-numbered) backups to keep and the number of newest (highest-numbered) ones to keep, each time a new backup is made. Recall that these values are used just after a new backup version is made; that newly made backup is included in the count in `kept-new-versions`. By default, both variables are 2.

If `trim-versions-without-asking` is non-`nil`, the excess middle versions are deleted without a murmur. If it is `nil`, the default, then you are asked whether the excess middle versions should really be deleted.

Dired's `.` (Period) command can also be used to delete old versions. See Section 15.7 [Dired], page 99.

15.3.1.3 Copying vs. Renaming

Backup files can be made by copying the old file or by renaming it. This makes a difference when the old file has multiple names. If the old file is renamed into the backup file, then the alternate names become names for the backup file. If the old file is copied instead, then the alternate names remain names for the file that you are editing, and the contents accessed by those names will be the new contents.

The method of making a backup file may also affect the file's owner and group. If copying is used, these do not change. If renaming is used, you become the file's owner, and the file's group becomes the default (different operating systems have different defaults for the group).

Having the owner change is usually a good idea, because then the owner always shows who last edited the file. Also, the owners of the backups show who produced those versions. Occasionally there is a file whose owner should not change; it is a good idea for such files to contain local variable lists to set `backup-by-copying-when-mismatch` for them alone (see Section 28.2.4 [File Variables], page 221).

The choice of renaming or copying is controlled by three variables. Normally, renaming is done. If the variable `backup-by-copying` is non-`nil`, copying is used. Otherwise, if the variable `backup-by-copying-when-linked` is non-`nil`, then copying is done for files that have multiple names, but renaming may still done when the file being edited has only one name. If the variable `backup-by-copying-when-mismatch` is non-`nil`, then copying is done if renaming would cause the file's owner or group to change.

15.3.2 Protection against Simultaneous Editing

Simultaneous editing occurs when two users visit the same file, both make changes, and then both save them. If nobody were informed that this was happening, whichever user saved first would later find that his changes were lost. On some systems, Emacs notices immediately when the second user starts to change the file, and issues an immediate warning. When this is not possible, or if the second user has gone on to change the file despite the warning, Emacs checks later when the file is saved, and issues a second warning when a user is about to overwrite a file containing another user's changes. If the editing user takes the proper corrective action at this point, he can prevent actual loss of work.

When you make the first modification in an Emacs buffer that is visiting a file, Emacs records that you have locked the file. (It does this by writing another file in a directory reserved for this purpose.) The lock is removed when you save the changes. The idea is that the file is locked whenever the buffer is modified. If you begin to modify the buffer while the visited file is locked by someone else, this constitutes a collision, and Emacs asks you what to do. It does this by calling the Lisp function `ask-user-about-lock`, which you can redefine for the sake of customization. The standard definition of this function asks you a question and accepts three possible answers:

`s` Steal the lock. Whoever was already changing the file loses the lock, and you gain the lock.

`p` Proceed. Go ahead and edit the file despite its being locked by someone else.

`q` Quit. This causes an error (`file-locked`) and the modification you were trying to make in the buffer does not actually take place.

Note that locking works on the basis of a file name; if a file has multiple names, Emacs does not realize that the two names are the same file and cannot prevent two user from editing it simultaneously under different names. However, basing locking on names means that Emacs can interlock the editing of new files that will not really exist until they are saved.

Some systems are not configured to allow Emacs to make locks. On these systems, Emacs cannot detect trouble in advance, but it still can detect it in time to prevent you from overwriting someone else's changes.

Every time Emacs saves a buffer, it first checks the last-modification date of the existing file on disk to see that it has not changed since the file was last visited or saved. If the date does not match, it implies that changes were made in the file in some other way, and these changes are about to be lost if Emacs actually does save. To prevent this, Emacs prints a warning message and asks for confirmation before saving. Occasionally you will know why the file was changed and know that it does not matter; then you can answer `yes` and proceed. Otherwise, you should cancel the save with `C-g` and investigate the situation.

The first thing you should do when notified that simultaneous editing has already taken place is to list the directory with `C-u C-x C-d` (see Section 15.6 [Directory Listing], page 98). This will show the file's current author. You should attempt to contact him to warn him not to continue editing. Often the next step is to save the contents of your Emacs buffer under a different name, and use `diff` to compare the two files.

Simultaneous editing checks are also made when you visit with `C-x C-f` a file that is already visited and when you start to modify a file. This is not strictly necessary, but it can cause you to find out about the problem earlier, when perhaps correction takes less work.

15.4 Reverting a Buffer

If you have made extensive changes to a file and then change your mind about them, you can get rid of them by reading in the previous version of the file. To do this, use `M-x revert-buffer`, which operates on the current buffer. Since this is a very dangerous thing to do, you must confirm it with `yes`.

If the current buffer has been auto-saved more recently than it has been saved for real, `revert-buffer` offers to read the auto save file instead of the visited file (see Section 15.5 [Auto Save], page 96). This question comes before the usual request for confirmation, and demands `y` or `n` as an answer. If you have started to type `yes` for confirmation without realizing that the other question was going to be asked, the `y` will answer that question, but the `es` will not be valid confirmation. So you will have a chance to cancel the operation with `C-g` and try it again with the answers that you really intend.

`revert-buffer` keeps point at the same distance (measured in characters) from the beginning of the file. If the file was edited only slightly, you will be at approximately the same piece of text after reverting as before. If you have made drastic changes, the same value of point in the old file may address a totally different piece of text.

A buffer reverted from its visited file is marked "not modified" until another change is made.

Some kinds of buffers whose contents reflect data bases other than files, such as Dired buffers, can also be reverted. For them, reverting means recalculating their contents from the appropriate data base. Buffers created randomly with `C-x b` cannot be reverted; `revert-buffer` reports an error when asked to do so.

15.5 Auto-Saving: Protection Against Disasters

Emacs saves all the visited files from time to time (based on counting your keystrokes) without being asked. This is called *auto-saving*. It prevents you from losing more than a limited amount of work if the system crashes.

When Emacs determines that it is time for auto-saving, each buffer is considered, and is auto-saved if auto-saving is turned on for it and it has been changed since the last time it was auto-saved. If any auto-saving is done, the message '`Auto-saving...`' is displayed in the echo area until auto-saving is finished. Errors occurring during auto-saving are caught so that they do not interfere with the execution of commands you have been typing.

15.5.1 Auto-Save Files

Auto-saving does not normally save in the files that you visited, because it can be very undesirable to save a program that is in an inconsistent state when you have made half of a planned change. Instead, auto-saving is done in a different file called the *auto-save file*, and the visited file is changed only when you request saving explicitly (such as with `C-x C-s`).

Normally, the auto-save file name is made by appending '`#`' to the front and rear of the visited file name. Thus, a buffer visiting file '`foo.c`' would be auto-saved in a file '`#foo.c#`'. Most buffers that are not visiting files are auto-saved only if you request it explicitly; when they are auto-saved, the auto-save file name is made by appending '`#%`' to the front and '`#`' to the rear of buffer name. For example, the '`*mail*`' buffer in which you compose messages to be sent is auto-saved in a file named '`#%*mail*#`'. Auto-save file names are made this way unless you reprogram parts of Emacs to do something different (the functions `make-auto-save-file-name` and `auto-save-file-name-p`). The file name to be used for auto-saving in a buffer is calculated when auto-saving is turned on in that buffer.

If you want auto-saving to be done in the visited file, set the variable `auto-save-visited-file-name` to be non-`nil`. In this mode, there is really no difference between auto-saving and explicit saving.

A buffer's auto-save file is deleted when you save the buffer in its visited file. To inhibit this, set the variable `delete-auto-save-files` to `nil`. Changing the visited file name with `C-x C-w` or `set-visited-file-name` renames any auto-save file to go with the new visited name.

15.5.2 Controlling Auto-Saving

Each time you visit a file, auto-saving is turned on for that file's buffer if the variable `auto-save-default` is non-`nil` (but not in batch mode; see Chapter 3 [Entering Emacs], page 21). The default for this variable is `t`, so auto-saving is the usual practice for file-visiting buffers. Auto-saving can be turned on or off for any existing buffer with the command `M-x auto-save-mode`. Like other minor mode commands, `M-x auto-save-mode` turns auto-saving on with a positive argument, off with a zero or negative argument; with no argument, it toggles.

Emacs does auto-saving periodically based on counting how many characters you have typed since the last time auto-saving was done. The variable `auto-save-interval` specifies how many characters there are between auto-saves. By default, it is 300. Emacs also auto-saves whenever you call the function `do-auto-save`.

Emacs also does auto-saving whenever it gets a fatal error. This includes killing the Emacs job with a shell command such as `kill %emacs`, or disconnecting a phone line or network connection.

15.5.3 Recovering Data from Auto-Saves

The way to use the contents of an auto-save file to recover from a loss of data is with the command `M-x recover-file RET` *file* `RET`. This visits *file* and then (after your confirmation) restores the contents from from its auto-save file '`#`*file*`#`'. You can then save with `C-x C-s` to put the recovered text into *file* itself. For example, to recover file '`foo.c`' from its auto-save file '`#foo.c#`', do:

```
M-x recover-file RET foo.c RET
C-x C-s
```

Before asking for confirmation, `M-x recover-file` displays a directory listing describing the specified file and the auto-save file, so you can compare their sizes and dates. If the auto-save file is older, `M-x recover-file` does not offer to read it.

Auto-saving is disabled by `M-x recover-file` because using this command implies that the auto-save file contains valuable data from a past session. If you save the data in the visited file and then go on to make new changes, you should turn auto-saving back on with `M-x auto-save-mode`.

15.6 Listing a File Directory

Files are classified by Unix into *directories*. A *directory listing* is a list of all the files in a directory. Emacs provides directory listings in brief format (file names only) and verbose format (sizes, dates, and authors included).

`C-x C-d` *dir-or-pattern*
: Print a brief directory listing (`list-directory`).

`C-u C-x C-d` *dir-or-pattern*
: Print a verbose directory listing.

The command to print a directory listing is `C-x C-d` (`list-directory`). It reads using the minibuffer a file name which is either a directory to be listed or a wildcard-containing pattern for the files to be listed. For example,

```
C-x C-d /u2/emacs/etc RET
```

lists all the files in directory '`/u2/emacs/etc`'. An example of specifying a file name pattern is

```
C-x C-d /u2/emacs/src/*.c RET
```

Normally, `C-x C-d` prints a brief directory listing containing just file names. A numeric argument (regardless of value) tells it to print a verbose listing (like `ls -l`).

The text of a directory listing is obtained by running `ls` in an inferior process. Two Emacs variables control the switches passed to `ls`: `list-directory-brief-switches` is a string giving the switches to use in brief listings (`"-CF"` by default), and `list-directory-verbose-switches` is a string giving the switches to use in a verbose listing (`"-l"` by default).

15.7 Dired, the Directory Editor

Dired makes it easy to delete or visit many of the files in a single directory at once. It makes an Emacs buffer containing a listing of the directory. You can use the normal Emacs commands to move around in this buffer, and special Dired commands to operate on the files.

15.7.1 Entering Dired

To invoke dired, do `C-x d` or `M-x dired`. The command reads a directory name or wildcard file name pattern as a minibuffer argument just like the `list-directory` command, `C-x C-d`. Where `dired` differs from `list-directory` is in naming the buffer after the directory name or the wildcard pattern used for the listing, and putting the buffer into Dired mode so that the special commands of Dired are available in it. The variable `dired-listing-switches` is a string used as an argument to `ls` in making the directory; this string *must* contain '`-l`'.

To display the Dired buffer in another window rather than in the selected window, use `C-x 4 d` (`dired-other-window`) instead of `C-x d`.

15.7.2 Editing in Dired

Once the Dired buffer exists, you can switch freely between it and other Emacs buffers. Whenever the Dired buffer is selected, certain special commands are provided that operate on files that are listed. The Dired buffer is "read-only", and inserting text in it is not useful, so ordinary printing characters such as `d` and `x` are used for Dired commands. Most Dired commands operate on the file described by the line that point is on. Some commands perform operations immediately; others "flag" the file to be operated on later.

Most Dired commands that operate on the current line's file also treat a numeric argument as a repeat count, meaning to act on the files of the next few lines. A negative argument means to operate on the files of the preceding lines, and leave point on the first of those lines.

All the usual Emacs cursor motion commands are available in Dired buffers. Some special purpose commands are also provided. The keys `C-n` and `C-p` are redefined so that they try to position the cursor at the beginning of the filename on the line, rather than at the beginning of the line.

For extra convenience, `SPC` and `n` in Dired are equivalent to `C-n`. `p` is equivalent to `C-p`. Moving by lines is done so often in Dired that it deserves to be easy to type. `DEL` (move up and unflag) is often useful simply for moving up.

The `g` command in Dired runs `revert-buffer` to reinitialize the buffer from the actual disk directory and show any changes made in the directory by programs other than Dired. All deletion flags in the Dired buffer are lost when this is done.

15.7.3 Deleting Files with Dired

The primary use of Dired is to flag files for deletion and then delete them.

`d` Flag this file for deletion.

`u` Remove deletion-flag on this line.

`DEL` Remove deletion-flag on previous line, moving point to that line.

`x` Delete the files that are flagged for deletion.

`#` Flag all auto-save files (files whose names start and end with '`#`') for deletion (see Section 15.5 [Auto Save], page 96).

`~` Flag all backup files (files whose names end with '`~`') for deletion (see Section 15.3.1 [Backup], page 92).

`.` (Period) Flag excess numeric backup files for deletion. The oldest and newest few backup files of any one file are exempt; the middle ones are flagged.

You can flag a file for deletion by moving to the line describing the file and typing `d` or `C-d`. The deletion flag is visible as a '`D`' at the beginning of the line. Point is moved to the beginning of the next line, so that repeated `d` commands flag successive files.

The files are flagged for deletion rather than deleted immediately to avoid the danger of deleting a file accidentally. Until you direct Dired to delete the flagged files, you can remove deletion flags using the commands `u` and `DEL`. `u` works just like `d`, but removes flags rather than making flags. `DEL` moves upward, removing flags; it is like `u` with numeric argument automatically negated.

To delete the flagged files, type `x`. This command first displays a list of all the file names flagged for deletion, and requests confirmation with `yes`. Once you confirm, all the flagged files are deleted, and their lines are deleted from the text of the Dired buffer. The shortened Dired buffer remains selected. If you answer `no` or quit with `C-g`, you return immediately to Dired, with the deletion flags still present and no files actually deleted.

The `#`, `~` and `.` commands flag many files for deletion, based on their names. These commands are useful precisely because they do not actually delete any files; you can remove the deletion flags from any flagged files that you really wish to keep.

`#` flags for deletion all files that appear to have been made by auto-saving (that is, files whose names begin and end with '`#`'). `~` flags for deletion all files that appear to have been made as backups for files that were edited (that is, files whose names end with '`~`').

`.` (Period) flags just some of the backup files for deletion: only numeric backups that are not among the oldest few nor the newest few backups of any one file. Normally `dired-kept-versions` (not `kept-new-versions`; that applies only when saving) specifies the number of newest versions of each file to keep, and `kept-old-versions` specifies the number of oldest versions to keep. Period with a positive numeric argument, as in `C-u 3 .`, specifies the number of newest versions to keep, overriding `dired-kept-versions`. A negative numeric argument overrides `kept-old-versions`, using minus the value of the argument to specify the number of oldest versions of each file to keep.

15.7.4 Immediate File Operations in Dired

Some file operations in Dired take place immediately when they are requested.

`c`
: Copies the file described on the current line. You must supply a file name to copy to, using the minibuffer.

`f`
: Visits the file described on the current line. It is just like typing `C-x C-f` and supplying that file name. If the file on this line is a subdirectory, `f` actually causes Dired to be invoked on that subdirectory. See Section 15.2 [Visiting], page 88.

`o`
: Like `f`, but uses another window to display the file's buffer. The Dired buffer remains visible in the first window. This is like using `C-x 4 C-f` to visit the file. See Chapter 17 [Windows], page 111.

`r` Renames the file described on the current line. You must supply a file name to rename to, using the minibuffer.

`v` Views the file described on this line using `M-x view-file`. Viewing a file is like visiting it, but is slanted toward moving around in the file conveniently and does not allow changing the file. See Section 15.8 [Misc File Ops], page 102. Viewing a file that is a directory runs Dired on that directory.

15.8 Miscellaneous File Operations

Emacs has commands for performing many other operations on files. All operate on one file; they do not accept wild card file names.

`M-x view-file` allows you to scan or read a file by sequential screenfuls. It reads a file name argument using the minibuffer. After reading the file into an Emacs buffer, `view-file` reads and displays one windowful. You can then type `SPC` to scroll forward one windowful, or `DEL` to scroll backward. Various other commands are provided for moving around in the file, but none for changing it; type `C-h` while viewing for a list of them. They are mostly the same as normal Emacs cursor motion commands. To exit from viewing, type `C-c`.

`M-x insert-file` inserts a copy of the contents of the specified file into the current buffer at point, leaving point unchanged before the contents and the mark after them. See Chapter 9 [Mark], page 49.

`M-x write-region` is the inverse of `M-x insert-file`; it copies the contents of the region into the specified file. `M-x append-to-file` adds the text of the region to the end of the specified file.

`M-x delete-file` deletes the specified file, like the `rm` command in the shell. If you are deleting many files in one directory, it may be more convenient to use Dired (see Section 15.7 [Dired], page 99).

`M-x rename-file` reads two file names *old* and *new* using the minibuffer, then renames file *old* as *new*. If a file named *new* already exists, you must confirm with `yes` or renaming is not done; this is because renaming causes the old meaning of the name *new* to be lost. If *old* and *new* are on different file systems, the file *old* is copied and deleted.

The similar command `M-x add-name-to-file` is used to add an additional name to an existing file without removing its old name. The new name must belong on the same file system that the file is on.

`M-x copy-file` reads the file *old* and writes a new file named *new* with the same contents. Confirmation is required if a file named *new* already exists, because copying has the consequence of overwriting the old contents of the file *new*.

`M-x make-symbolic-link` reads two file names *old* and *linkname*, and then creates a symbolic link named *linkname* and pointing at *old*. The effect is that future attempts to open file *linkname* will refer to whatever file is named *old* at the time the opening is done, or will get an error if the name *old* is not in use at that time. Confirmation is required when creating the link if *linkname* is in use. Note that not all systems support symbolic links.

16 Using Multiple Buffers

The text you are editing in Emacs resides in an object called a *buffer*. Each time you visit a file, a buffer is created to hold the file's text. Each time you invoke Dired, a buffer is created to hold the directory listing. If you send a message with `C-x m`, a buffer named '`*mail*`' is used to hold the text of the message. When you ask for a command's documentation, that appears in a buffer called '`*Help*`'.

At any time, one and only one buffer is *selected*. It is also called the *current buffer*. Often we say that a command operates on "the buffer" as if there were only one; but really this means that the command operates on the selected buffer (most commands do).

When Emacs makes multiple windows, each window has a chosen buffer which is displayed there, but at any time only one of the windows is selected and its chosen buffer is the selected buffer. Each window's mode line displays the name of the buffer that the window is displaying (see Chapter 17 [Windows], page 111).

Each buffer has a name, which can be of any length, and you can select any buffer by giving its name. Most buffers are made by visiting files, and their names are derived from the files' names. But you can also create an empty buffer with any name you want. A newly started Emacs has a buffer named '`*scratch*`' which can be used for evaluating Lisp expressions in Emacs. The distinction between upper and lower case matters in buffer names.

Each buffer records individually what file it is visiting, whether it is modified, and what major mode and minor modes are in effect in it (see Chapter 18 [Major Modes], page 115). Any Emacs variable can be made *local to* a particular buffer, meaning its value in that buffer can be different from the value in other buffers. See Section 28.2.3 [Locals], page 220.

16.1 Creating and Selecting Buffers

`C-x b` *buffer* `RET`
: Select or create a buffer named *buffer* (`switch-to-buffer`).

`C-x 4 b` *buffer* `RET`
: Similar but select a buffer named *buffer* in another window (`switch-to-buffer-other-window`).

To select the buffer named *bufname*, type `C-x b` *bufname* `RET`. This is the command `switch-`

`to-buffer` with argument *bufname*. You can use completion on an abbreviation for the buffer name you want (see Section 6.3.2 [Completion], page 38). An empty argument to `C-x b` specifies the most recently selected buffer that is not displayed in any window.

Most buffers are created by visiting files, or by Emacs commands that want to display some text, but you can also create a buffer explicitly by typing `C-x b` *bufname* `RET`. This makes a new, empty buffer which is not visiting any file, and selects it for editing. Such buffers are used for making notes to yourself. If you try to save one, you are asked for the file name to use. The new buffer's major mode is determined by the value of `default-major-mode` (see Chapter 18 [Major Modes], page 115).

Note that `C-x C-f`, and any other command for visiting a file, can also be used to switch buffers. See Section 15.2 [Visiting], page 88.

16.2 Listing Existing Buffers

`C-x C-b` List the existing buffers (`list-buffers`).

To print a list of all the buffers that exist, type `C-x C-b`. Each line in the list shows one buffer's name, major mode and visited file. '`*`' at the beginning of a line indicates the buffer is "modified". If several buffers are modified, it may be time to save some with `C-x s` (see Section 15.3 [Saving], page 90). '`%`' indicates a read-only buffer. '`.`' marks the selected buffer. Here is an example of a buffer list:

```
 MR Buffer         Size  Mode           File
 -- ------         ----  ----           ----
.*  emacs.tex      383402 Texinfo       /u2/emacs/man/emacs.tex
    *Help*         1287  Fundamental
    files.el       23076 Emacs-Lisp     /u2/emacs/lisp/files.el
  % RMAIL          64042 RMAIL          /u/rms/RMAIL
 *% man            747   Dired
    net.emacs      343885 Fundamental   /u/rms/net.emacs
    fileio.c       27691 C              /u2/emacs/src/fileio.c
    NEWS           67340 Text           /u2/emacs/etc/NEWS
    *scratch*    0  Lisp Interaction
```

Note that the buffer '`*Help*`' was made by a help request; it is not visiting any file. The buffer `man` was made by Dired on the directory '`/u2/emacs/man/`'.

16.3 Miscellaneous Buffer Operations

`C-x C-q` Toggle read-only status of buffer (`toggle-read-only`).

`M-x rename-buffer`
: Change the name of the current buffer.

`M-x view-buffer`
: Scroll through a buffer.

A buffer can be *read-only*, which means that commands to change its text are not allowed. Normally, read-only buffers are made by subsystems such as Dired and Rmail that have special commands to operate on the text; a read-only buffer is also made if you visit a file that is protected so you cannot write it. If you wish to make changes in a read-only buffer, use the command `C-x C-q` (`toggle-read-only`). It makes a read-only buffer writable, and makes a writable buffer read-only. This works by setting the variable `buffer-read-only`, which has a local value in each buffer and makes the buffer read-only if its value is non-`nil`.

`M-x rename-buffer` changes the name of the current buffer. Specify the new name as a minibuffer argument. There is no default. If you specify a name that is in use for some other buffer, an error happens and no renaming is done.

`M-x view-buffer` is much like `M-x view-file` (see Section 15.8 [Misc File Ops], page 102) except that it examines an already existing Emacs buffer. View mode provides commands for scrolling through the buffer conveniently but not for changing it. When you exit View mode, the value of point that resulted from your perusal remains in effect.

The commands `C-x a` (`append-to-buffer`) and `M-x insert-buffer` can be used to copy text from one buffer to another. See Section 10.3 [Accumulating Text], page 58.

16.4 Killing Buffers

After you use Emacs for a while, you may accumulate a large number of buffers. You may then find it convenient to eliminate the ones you no longer need. There are several commands provided for doing this.

`C-x k` Kill a buffer, specified by name (`kill-buffer`).

`M-x kill-some-buffers`
: Offer to kill each buffer, one by one.

C-x k (kill-buffer) kills one buffer, whose name you specify in the minibuffer. The default, used if you type just RET in the minibuffer, is to kill the current buffer. If the current buffer is killed, another buffer is selected; a buffer that has been selected recently but does not appear in any window now is chosen to be selected. If the buffer being killed is modified (has unsaved editing) then you are asked to confirm with yes before the buffer is killed.

The command M-x kill-some-buffers asks about each buffer, one by one. An answer of y means to kill the buffer. Killing the current buffer or a buffer containing unsaved changes selects a new buffer or asks for confirmation just like kill-buffer.

16.5 Operating on Several Buffers

The *buffer-menu* facility is like a "Dired for buffers"; it allows you to request operations on various Emacs buffers by editing an Emacs buffer containing a list of them. You can save buffers, kill them (here called *deleting* them, for consistency with Dired), or display them.

M-x buffer-menu
: Begin editing a buffer listing all Emacs buffers.

The command buffer-menu writes a list of all Emacs buffers into the buffer '*Buffer List*', and selects that buffer in Buffer Menu mode. The buffer is read-only, and can only be changed through the special commands described in this section. Most of these commands are graphic characters. The usual Emacs cursor motion commands can be used in the '*Buffer List*' buffer. The following special commands apply to the buffer described on the current line.

d
: Request to delete (kill) the buffer, then move down. The request shows as a 'D' on the line, before the buffer name. Requested deletions take place when the x command is used.

k
: Synonym for d.

C-d
: Like d but move up afterwards instead of down.

s
: Request to save the buffer. The request shows as an 'S' on the line. Requested saves take place when the x command is used. You may request both saving and deletion for the same buffer.

~
: Mark buffer "unmodified". The command ~ does this immediately when typed.

x
: Perform previously requested deletions and saves.

u
: Remove any request made for the current line, and move down.

DEL
: Move to previous line and remove any request made for that line.

All the commands that put in or remove flags to request later operations also move down a line, and accept a numeric argument as a repeat count, unless otherwise specified.

There are also special commands to use the buffer list to select another buffer, and to specify one or more other buffers for display in additional windows.

`1` Select the buffer in a full-screen window. This command takes effect immediately.

`2` Immediately set up two windows, with this buffer in one, and the previously selected buffer (aside from the buffer '`*Buffer List*`') in the other.

`f` Immediately select the buffer in place of the '`*Buffer List*`' buffer.

`o` Immediately select the buffer in another window as if by `C-x 4 b`, leaving '`*Buffer List*`' visible.

`q` Immediately select this buffer, and also display in other windows any buffers previously flagged with the `m` command. If there are no such buffers, this command is equivalent to `1`.

`m` Flag this buffer to be displayed in another window if the `q` command is used. The request shows as a '`>`' at the beginning of the line. The same buffer may not have both a delete request and a display request.

All that `buffer-menu` does directly is create and select a suitable buffer, and turn on Buffer Menu mode. Everything else described above is implemented by the special commands provided in Buffer Menu mode. One consequence of this is that you can switch from the '`*Buffer List*`' buffer to another Emacs buffer, and edit there. You can reselect the `buffer-menu` buffer later, to perform the operations already requested, or you can kill it, or pay no further attention to it.

The only difference between `buffer-menu` and `list-buffers` is that `buffer-menu` selects the '`*Buffer List*`' buffer and `list-buffers` does not. If you run `list-buffers` (that is, type `C-x C-b`) and select the buffer list manually, you can use all of the commands described here.

17 Multiple Windows

Emacs can split the screen into two or many windows, which can display parts of different buffers, or different parts of one buffer.

17.1 Concepts of Emacs Windows

When multiple windows are being displayed, each window has an Emacs buffer designated for display in it. The same buffer may appear in more than one window; if it does, any changes in its text are displayed in all the windows where it appears. But the windows showing the same buffer can show different parts of it, because each window has its own value of point.

At any time, one of the windows is the *selected window*; the buffer this window is displaying is the current buffer. The terminal's cursor shows the location of point in this window. Each other window has a location of point as well, but since the terminal has only one cursor there is no way to show where those locations are.

Commands to move point affect the value of point for the selected Emacs window only. They do not change the value of point in any other Emacs window, even one showing the same buffer. The same is true for commands such as `C-x b` to change the selected buffer in the selected window; they do not affect other windows at all. However, there are other commands such as `C-x 4 b` that select a different window and switch buffers in it. Also, all commands that display information in a window, including (for example) `C-h f` (`describe-function`) and `C-x C-b` (`list-buffers`), work by switching buffers in a nonselected window without affecting the selected window.

Each window has its own mode line, which displays the buffer name, modification status and major and minor modes of the buffer that is displayed in the window. See Chapter 2 [Mode Line], page 17, for full details on the mode line.

17.2 Splitting Windows

`C-x 2` Split the selected window into two windows, one above the other (`split-window-vertically`).

`C-x 5` Split the selected window into two windows positioned side by side (`split-window-horizontally`).

The command `C-x 2` (`split-window-vertically`) breaks the selected window into two windows, one above the other. Both windows start out displaying the same buffer, with the same value of point. By default the two windows each get half the height of the window that was split; a numeric argument specifies how many lines to give to the top window.

`C-x 5` (`split-window-horizontally`) breaks the selected window into two side-by-side windows. A numeric argument specifies how many columns to give the one on the left. A line of vertical bars separates the two windows. Windows that are not the full width of the screen have mode lines, but they are truncated; also, they do not always appear in inverse video, because, the Emacs display routines have not been taught how to display a region of inverse video that is only part of a line on the screen.

When a window is less than the full width, text lines too long to fit are frequent. Continuing all those lines might be confusing. The variable `truncate-partial-width-windows` can be set non-`nil` to force truncation in all windows less than the full width of the screen, independent of the buffer being displayed and its value for `truncate-lines`. See Section 4.7 [Continuation Lines], page 29.

Horizontal scrolling is often used in side-by-side windows. See Chapter 12 [Display], page 65.

17.3 Using Other Windows

`C-x o` Select another window (`other-window`). That is `o`, not zero.

`C-M-v` Scroll the next window (`scroll-other-window`).

`M-x compare-windows`
: Find next place where the text in the selected window does not match the text in the next window.

To select a different window, use `C-x o` (`other-window`). That is an `o`, for 'other', not a zero. When there are more than two windows, this command moves through all the windows in a cyclic order, generally top to bottom and left to right. From the rightmost and bottommost window, it goes back to the one at the upper left corner. A numeric argument means to move several steps in the cyclic order of windows. A negative argument moves around the cycle in the opposite order. When the minibuffer is active, the minibuffer is the last window in the cycle; you can switch from the minibuffer window to one of the other windows, and later switch back and finish supplying the minibuffer argument that is requested. See Section 6.2 [Minibuffer Edit], page 36.

The usual scrolling commands (see Chapter 12 [Display], page 65) apply to the selected window only, but there is one command to scroll the next window. `C-M-v` (`scroll-other-window`) scrolls the window that `C-x o` would select. It takes arguments, positive and negative, like `C-v`.

The command `M-x compare-windows` compares the text in the current window with that in the next window. Comparison starts at point in each window. Point moves forward in each window, a character at a time in each window, until the next characters in the two windows are different. Then the command is finished.

17.4 Displaying in Another Window

`C-x 4` is a prefix key for commands that select another window (splitting the window if there is only one) and select a buffer in that window. Different `C-x 4` commands have different ways of finding the buffer to select.

`C-x 4 b` *bufname* `RET`
: Select buffer *bufname* in another window. This runs `switch-to-buffer-other-window`.

`C-x 4 f` *filename* `RET`
: Visit file *filename* and select its buffer in another window. This runs `find-file-other-window`. See Section 15.2 [Visiting], page 88.

`C-x 4 d` *directory* `RET`
: Select a Dired buffer for directory *directory* in another window. This runs `dired-other-window`. See Section 15.7 [Dired], page 99.

`C-x 4 m`
: Start composing a mail message in another window. This runs `mail-other-window`, and its same-window version is `C-x m` (see Chapter 25 [Sending Mail], page 187).

`C-x 4 .`
: Find a tag in the current tag table in another window. This runs `find-tag-other-window`, the multiple-window variant of `M-.`. (see Section 21.11 [Tags], page 155).

17.5 Deleting and Rearranging Windows

`C-x 0`
: Get rid of the selected window (`kill-window`). That is a zero.

`C-x 1`
: Get rid of all windows except the selected one (`delete-other-windows`).

`C-x ^`
: Make the selected window taller, at the expense of the other(s) (`enlarge-window`).

`C-x }`
: Make the selected window wider (`enlarge-window-horizontally`).

To delete a window, type `C-x 0` (`delete-window`). (That is a zero.) The space occupied by the deleted window is distributed among the other active windows (but not the minibuffer window, even if that is active at the time). Once a window is deleted, its attributes are forgotten; there is no automatic way to make another window of the same shape or showing the same buffer. But the buffer continues to exist, and you can select it in any window with `C-x b`.

`C-x 1` (`delete-other-windows`) is more powerful than `C-x 0`; it deletes all the windows except the selected one (and the minibuffer); the selected window expands to use the whole screen except for the echo area.

To readjust the division of space among existing windows, use `C-x ^` (`enlarge-window`). It makes the currently selected window get one line bigger, or as many lines as is specified with a numeric argument. With a negative argument, it makes the selected window smaller. `C-x }` (`enlarge-window-horizontally`) makes the selected window wider by the specified number of columns. The extra screen space given to a window comes from one of its neighbors, if that is possible; otherwise, all the competing windows are shrunk in the same proportion. If this makes any windows too small, those windows are deleted and their space is divided up. The minimum size is specified by the variables `window-min-height` and `window-min-width`.

18 Major Modes

Emacs has many different *major modes*, each of which customizes Emacs for editing text of a particular sort. The major modes are mutually exclusive, and each buffer has one major mode at any time. The mode line normally contains the name of the current major mode, in parentheses. See Chapter 2 [Mode Line], page 17.

The least specialized major mode is called *Fundamental mode*. This mode has no mode-specific redefinitions or variable settings, so that each Emacs command behaves in its most general manner, and each option is in its default state. For editing any specific type of text, such as Lisp code or English text, you should switch to the appropriate major mode, such as Lisp mode or Text mode.

Selecting a major mode changes the meanings of a few keys to become more specifically adapted to the language being edited. The ones which are changed frequently are `TAB`, `DEL`, and `LFD`. In addition, the commands which handle comments use the mode to determine how comments are to be delimited. Many major modes redefine the syntactical properties of characters appearing in the buffer. See Section 28.5 [Syntax], page 229.

The major modes fall into three major groups. Lisp mode (which has several variants), C mode and Muddle mode are for specific programming languages. Text mode, Nroff mode, TeX mode and Outline mode are for editing English text. The remaining major modes are not intended for use on users' files; they are used in buffers created for specific purposes by Emacs, such as Dired mode for buffers made by Dired (see Section 15.7 [Dired], page 99), and Mail mode for buffers made by `C-x m` (see Chapter 25 [Sending Mail], page 187), and Shell mode for buffers used for communicating with an inferior shell process (see Section 27.4.2 [Interactive Shell], page 212).

Most programming language major modes specify that only blank lines separate paragraphs. This is so that the paragraph commands remain useful. See Section 20.4 [Paragraphs], page 132. They also cause Auto Fill mode to use the definition of `TAB` to indent the new lines it creates. This is because most lines in a program are usually indented. See Chapter 19 [Indentation], page 117.

18.1 How Major Modes are Chosen

You can select a major mode explicitly for the current buffer, but most of the time Emacs determines which mode to use based on the file name or some text in the file.

Explicit selection of a new major mode is done with a `M-x` command. From the name of a major mode, add `-mode` to get the name of a command to select that mode. Thus, you can enter Lisp mode by executing `M-x lisp-mode`.

When you visit a file, Emacs usually chooses the right major mode based on the file's name. For example, files whose names end in `.c` are edited in C mode. The correspondence between file names and major mode is controlled by the variable `auto-mode-alist`. Its value is a list in which each element has the form

```
(regexp . mode-function)
```

For example, one element normally found in the list has the form `("\\.c$" . c-mode)`, and it is responsible for selecting C mode for files whose names end in '`.c`'. (Note that '`\\`' is needed in Lisp syntax to include a '`\`' in the string, which is needed to suppress the special meaning of '`.`' in regexps.) The only practical way to change this variable is with Lisp code.

You can specify which major mode should be used for editing a certain file by a special sort of text in the first nonblank line of the file. The mode name should appear in this line both preceded and followed by '`-*-`'. Other text may appear on the line as well. For example,

```
;-*-Lisp-*-
```

tells Emacs to use Lisp mode. Note how the semicolon is used to make Lisp treat this line as a comment. Such an explicit specification overrides any defaulting based on the file name.

Another format of mode specification is

```
-*-Mode: modename;-*-
```

which allows other things besides the major mode name to be specified. However, Emacs does not look for anything except the mode name.

The major mode can also be specified in a local variables list. See Section 28.2.4 [File Variables], page 221.

When a file is visited that does not specify a major mode to use, or when a new buffer is created with `C-x b`, the major mode used is that specified by the variable `default-major-mode`. Normally this value is the symbol `fundamental-mode`, which specifies Fundamental mode. If `default-major-mode` is `nil`, the major mode is taken from the previously selected buffer.

19 Indentation

`TAB` Indent current line "appropriately" in a mode-dependent fashion.

`LFD` Perform `RET` followed by `TAB` (`newline-and-indent`).

`M-^` Merge two lines (`delete-indentation`). This would cancel out the effect of `LFD`.

`C-M-o` Split line at point; text on the line after point becomes a new line indented to the same column that it now starts in (`split-line`).

`M-m` Move (forward or back) to the first nonblank character on the current line (`back-to-indentation`).

`C-M-\` Indent several lines to same column (`indent-region`).

`C-x TAB` Shift block of lines rigidly right or left (`indent-rigidly`).

`M-i` Indent from point to the next prespecified tab stop column (`tab-to-tab-stop`).

`M-x indent-relative`
Indent from point to under an indentation point in the previous line.

Most programming languages have some indentation convention. For Lisp code, lines are indented according to their nesting in parentheses. The same general idea is used for C code, though many details are different.

Whatever the language, to indent a line, use the `TAB` command. Each major mode defines this command to perform the sort of indentation appropriate for the particular language. In Lisp mode, `TAB` aligns the line according to its depth in parentheses. No matter where in the line you are when you type `TAB`, it aligns the line as a whole. In C mode, `TAB` implements a subtle and sophisticated indentation style that knows about many aspects of C syntax.

In Text mode, `TAB` runs the command `tab-to-tab-stop`, which indents to the next tab stop column. You can set the tab stops with `M-x edit-tab-stops`.

19.1 Indentation Commands and Techniques

If you just want to insert a tab character in the buffer, you can type `C-q TAB`.

To move over the indentation on a line, do `Meta-m` (`back-to-indentation`). This command, given anywhere on a line, positions point at the first nonblank character on the line.

To insert an indented line before the current line, do `C-a C-o TAB`. To make an indented line after the current line, use `C-e LFD`.

`C-M-o` (`split-line`) moves the text from point to the end of the line vertically down, so that the current line becomes two lines. `C-M-o` first moves point forward over any spaces and tabs. Then it inserts after point a newline and enough indentation to reach the same column point is on. Point remains before the inserted newline; in this regard, `C-M-o` resembles `C-o`.

To join two lines cleanly, use the `Meta-^` (`delete-indentation`) command to delete the indentation at the front of the current line, and the line boundary as well. They are replaced by a single space, or by no space if at the beginning of a line or before a '`)`' or after a '`(`'. To delete just the indentation of a line, go to the beginning of the line and use `Meta-\` (`delete-horizontal-space`), which deletes all spaces and tabs around the cursor.

There are also commands for changing the indentation of several lines at once. `Control-Meta-\` (`indent-region`) gives each line which begins in the region the "usual" indentation by invoking `TAB` at the beginning of the line. A numeric argument specifies the column to indent to, and each line is shifted left or right so that its first nonblank character appears in that column. `C-x TAB` (`indent-rigidly`) moves all of the lines in the region right by its argument (left, for negative arguments). The whole group of lines moves rigidly sideways, which is how the command gets its name.

`M-x indent-relative` indents at point based on the previous line (actually, the last nonempty line.) It inserts whitespace at point, moving point, until it is underneath an indentation point in the previous line. An indentation point is the end of a sequence of whitespace or the end of the line. If point is farther right than any indentation point in the previous line, the whitespace before point is deleted and the first indentation point then applicable is used. If no indentation point is applicable even then, `tab-to-tab-stop` is run (see next section).

`indent-relative` is the definition of `TAB` in Indented Text mode. See Chapter 20 [Text], page 121.

19.2 Tab Stops

For typing in tables, you can use Text mode's definition of `TAB`, `tab-to-tab-stop`. This command inserts indentation before point, enough to reach the next tab stop column. If you are not in Text mode, this function can be found on `M-i` anyway.

The tab stops used by `M-i` can be set arbitrarily by the user. They are stored in a variable called `tab-stop-list`, as a list of column-numbers in increasing order.

The convenient way to set the tab stops is using `M-x edit-tab-stops`, which creates and selects a buffer containing a description of the tab stop settings. You can edit this buffer to specify different tab stops, and then type `C-c C-c` to make those new tab stops take effect. In the tab stop buffer, `C-c C-c` runs the function `edit-tab-stops-note-changes` rather than its usual definition `save-buffer`. `edit-tab-stops` records which buffer was current when you invoked it, and stores the tab stops back in that buffer; normally all buffers share the same tab stops and changing them in one buffer affects all, but if you happen to make `tab-stop-list` local in one buffer then `edit-tab-stops` in that buffer will edit the local settings.

Here is what the text representing the tab stops looks like for ordinary tab stops every eight columns.

```
        :       :       :       :       :       :
0         1         2         3         4
0123456789012345678901234567890123456789012345678
To install changes, type C-c C-c
```

The first line contains a colon at each tab stop. The remaining lines are present just to help you see where the colons are and know what to do.

Note that the tab stops that control `tab-to-tab-stop` have nothing to do with displaying tab characters in the buffer. See Section 12.4 [Display Vars], page 67, for more information on that.

19.3 Tabs vs. Spaces

Emacs normally uses both tabs and spaces to indent lines. If you prefer, all indentation can be made from spaces only. To request this, set `indent-tabs-mode` to `nil`. This is a per-buffer variable; altering the variable affects only the current buffer, but there is a default value which you can change as well. See Section 28.2.3 [Locals], page 220.

There are also commands to convert tabs to spaces or vice versa, always preserving the columns of all nonblank text. `M-x tabify` scans the region for sequences of spaces, and converts sequences of at least three spaces to tabs if that can be done without changing indentation. `M-x untabify` changes all tabs in the region to appropriate numbers of spaces.

20 Commands for Human Languages

The term *text* has two widespread meanings in our area of the computer field. One is data that is a sequence of characters. Any file that you edit with Emacs is text, in this sense of the word. The other meaning is more restrictive: a sequence of characters in a human language for humans to read (possibly after processing by a text formatter), as opposed to a program or commands for a program.

Human languages have syntactic/stylistic conventions that can be supported or used to advantage by editor commands: conventions involving words, sentences, paragraphs, and capital letters. This chapter describes Emacs commands for all of these things. There are also commands for *filling*, or rearranging paragraphs into lines of approximately equal length. The commands for moving over and killing words, sentences and paragraphs, while intended primarily for editing text, are also often useful for editing programs.

Emacs has several major modes for editing human language text. If the file contains text pure and simple, use Text mode, which customizes Emacs in small ways for the syntactic conventions of text. For text which contains embedded commands for text formatters, Emacs has other major modes, each for a particular text formatter. Thus, for input to TeX, you would use TeX mode; for input to nroff, Nroff mode.

20.1 Text Mode

Editing files of text in a human language ought to be done using Text mode rather than Lisp or Fundamental mode. Invoke `M-x text-mode` to enter Text mode. In Text mode, `TAB` runs the function `tab-to-tab-stop`, which allows you to use arbitrary tab stops set with `M-x edit-tab-stops` (see Section 19.2 [Tab Stops], page 118). Features concerned with comments in programs are turned off except when explicitly invoked. The syntax table is changed so that periods are not considered part of a word, while apostrophes, backspaces and underlines are.

A similar variant mode is Indented Text mode, intended for editing text in which most lines are indented. This mode defines `TAB` to run `indent-relative` (see Chapter 19 [Indentation], page 117), and makes Auto Fill indent the lines it creates. The result is that normally a line made by Auto Filling, or by `LFD`, is indented just like the previous line. Use `M-x indented-text-mode` to select this mode.

Entering Text mode or Indented Text mode calls with no arguments the value of the variable `text-mode-hook`, if that value exists and is not `nil`. This value is also called when modes related to Text mode are entered; this includes Nroff mode, TeX mode, Outline mode and Mail mode. Your hook can look at the value of `major-mode` to see which of these modes is actually being entered.

20.1.1 Nroff Mode

Nroff mode is a mode like Text mode but modified to handle nroff commands present in the text. Invoke `M-x nroff-mode` to enter this mode. It differs from Text mode in only a few ways. All nroff command lines are considered paragraph separators, so that filling will never garble the nroff commands. Pages are separated by '`.bp`' commands. Comments start with backslash-doublequote. Also, three special commands are provided that are not in Text mode:

`M-n`
: Move to the beginning of the next line that isn't an nroff command (`forward-text-line`). An argument is a repeat count.

`M-p`
: Like `M-n` but move up (`backward-text-line`).

`M-?`
: Prints in the echo area the number of text lines (lines that are not nroff commands) in the region (`count-text-lines`).

The other feature of Nroff mode is that you can turn on Electric Nroff newline mode. This is a minor mode that you can turn on or off with `M-x electric-nroff-mode` (see Section 28.1 [Minor Modes], page 217). When the mode is on, each time you use `RET` to end a line that contains an nroff command that opens a kind of grouping, the matching nroff command to close that grouping is automatically inserted on the following line. For example, if you are at the beginning of a line and type `. ( b RET`, the matching command '`.)b`' will be inserted on a new line following point.

Entering Nroff mode calls with no arguments the value of the variable `text-mode-hook`, if that value exists and is not `nil`; then it does the same with the variable `nroff-mode-hook`.

20.1.2 TeX Mode

TeX is a powerful text formatter written by Donald Knuth; it is also free, like GNU Emacs. LaTeX is a simplified input format for TeX, implemented by TeX macros. It comes with TeX.

Emacs has a special TeX mode for editing TeX input files. It provides facilities for checking the balance of delimiters and for invoking TeX on all or part of the file.

TeX mode has two variants, Plain TeX mode and LaTeX mode (actually two distinct major modes which differ only slightly). They are designed for editing the two different input formats. The command `M-x tex-mode` looks at the contents of the buffer to determine whether the contents appear to be LaTeX input or not; it then selects the appropriate mode. If it can't tell which is right (e.g., the buffer is empty), the variable `TeX-default-mode` controls which mode is used.

The commands `M-x plain-tex-mode` and `M-x latex-mode` explicitly select the two variants of TeX mode. Use these commands when `M-x tex-mode` does not guess right.

TeX for Unix systems can be obtained from the University of Washington for a distribution fee.

To order a full distribution, send $140.00 for a 1/2 inch 9-track tape, $165.00 for two 4-track 1/4 inch cartridge tapes (foreign sites $150.00, for 1/2 inch, $175.00 for 1/4 inch, to cover the extra postage) payable to the University of Washington to:

> The Director
> Northwest Computer Support Group, DW-10
> University of Washington
> Seattle, Washington 98195

Purchase orders are acceptable, but there is an extra charge of $10.00, to pay for processing charges. (Total of $150 for domestic sites, $175 for foreign sites).

The normal distribution is a tar tape, blocked 20, 1600 bpi, on an industry standard 2400 foot half-inch reel. The physical format for the 1/4 inch streamer cartridges uses QIC-11, 8000 bpi, 4-track serpentine recording for the SUN. Also, SystemV tapes can be written in cpio format, blocked 5120 bytes, ASCII headers.

20.1.2.1 TeX Editing Commands

Here are the special commands provided in TeX mode for editing the text of the file.

`"`
: Insert, according to context, either ‘‘‘’ or ‘"’ or ‘’’’ (`TeX-insert-quote`).

`LFD`
: Insert a paragraph break (two newlines) and check the previous paragraph for unbalanced braces or dollar signs (`TeX-terminate-paragraph`).

`M-x validate-TeX-buffer`
: Check each paragraph in the buffer for unbalanced braces or dollar signs.

`M-{`
: Insert ‘`{}`’ and position point between them (`TeX-insert-braces`).

`M-}` Move forward past the next unmatched close brace (`up-list`).

`C-c C-f` Close a block for LaTeX (`TeX-close-LaTeX-block`).

In TeX, the character ‘"’ is not normally used; one uses ‘‘‘’ to start a quotation and ‘’’’ to end one. TeX mode defines the key " to insert ‘‘‘’ after whitespace or an open brace, ‘"’ after a backslash, or ‘’’’ otherwise. This is done by the command `TeX-insert-quote`. If you need the character ‘"’ itself in unusual contexts, use `C-q` to insert it. Also, " with a numeric argument always inserts that number of ‘"’ characters.

In TeX mode, ‘`$`’ has a special syntax code which attempts to understand the way TeX math mode delimiters match. When you insert a ‘`$`’ that is meant to exit math mode, the position of the matching ‘`$`’ that entered math mode is displayed for a second. This is the same feature that displays the open brace that matches a close brace that is inserted. However, there is no way to tell whether a ‘`$`’ enters math mode or leaves it; so when you insert a ‘`$`’ that enters math mode, the previous ‘`$`’ position is shown as if it were a match, even though they are actually unrelated.

If you prefer to keep braces balanced at all times, you can use `M-{` (`TeX-insert-braces`) to insert a pair of braces. It leaves point between the two braces so you can insert the text that belongs inside. Afterward, use the command `M-}` (`up-list`) to move forward past the close brace.

There are two commands for checking the matching of braces. `LFD` (`TeX-terminate-paragraph`) checks the paragraph before point, and inserts two newlines to start a new paragraph. It prints a message in the echo area if any mismatch is found. `M-x validate-TeX-buffer` checks the entire buffer, paragraph by paragraph. When it finds a paragraph that contains a mismatch, it displays point at the beginning of the paragraph for a few seconds and pushes a mark at that spot. Scanning continues until the whole buffer has been checked or until you type another key. The positions of the last several paragraphs with mismatches can be found in the mark ring (see Chapter 10 [Mark Ring], page 53).

Note that square brackets and parentheses are matched in TeX mode, not just braces. This is wrong for the purpose of checking TeX syntax. However, parentheses and square brackets are likely to be used in text as matching delimiters and it is useful for the various motion commands and automatic match display to work with them.

In LaTeX input, ‘`\begin`’ and ‘`\end`’ commands must balance. After you insert a ‘`\begin`’, use `C-c C-f` (`TeX-close-LaTeX-block`) to insert automatically a matching ‘`\end`’ (on a new line following the ‘`\begin`’). A blank line is inserted between the two, and point is left there.

20.1.2.2 TeX Printing Commands

You can invoke TeX as an inferior of Emacs on either the entire contents of the buffer or just a region at a time. Running TeX in this way on just one chapter is a good way to see what your changes look like without taking the time to format the entire file.

`C-c C-r` Invoke TeX on the current region, plus the buffer's header (`TeX-region`).

`C-c C-b` Invoke TeX on the entire current buffer (`TeX-buffer`).

`C-c C-l` Recenter the window showing output from the inferior TeX so that the last line can be seen (`TeX-recenter-output-buffer`).

`C-c C-k` Kill the inferior TeX (`TeX-kill-job`).

`C-c C-p` Print the output from the last `C-c C-r` or `C-c C-b` command (`TeX-print`).

`C-c C-q` Show the printer queue (`TeX-show-print-queue`).

You can pass the current buffer through an inferior TeX by means of `C-c C-b` (`TeX-buffer`). The formatted output appears in a file in '`/tmp`'; to print it, type `C-c C-p` (`TeX-print`). Afterward use `C-c C-q` (`TeX-show-print-queue`) to view the progress of your output towards being printed.

The console output from TeX, including any error messages, appear in a buffer called '`*TeX-shell*`'. If TeX gets an error, you can switch to this buffer and feed it input (this works as in Shell mode; see Section 27.4.2 [Interactive Shell], page 212). Without switching to this buffer you can scroll it so that its last line is visible by typing `C-c C-l`.

Type `C-c C-k` (`TeX-kill-job`) to kill the TeX process if you see that its output is no longer useful. Using `C-c C-b` or `C-c C-r` also kills any TeX process still running.

You can also pass an arbitrary region through an inferior TeX by typing `C-c C-r` (`TeX-region`). This is tricky, however, because most files of TeX input contain commands at the beginning to set parameters and define macros, without which no later part of the file will format correctly. To solve this problem, `C-c C-r` allows you to designate a part of the file as containing essential commands; it is included before the specified region as part of the input to TeX. The designated part of the file is called the *header*.

To indicate the bounds of the header in Plain TeX mode, you insert two special strings in the file. Insert '`%**start of header`' before the header, and '`%**end of header`' after it. Each string must appear entirely on one line, but there may be other text on the line before or after. The lines containing the two strings are included in the header. If '`%**start of header`' does not appear within the first 100 lines of the buffer, `C-c C-r` assumes that there is no header.

In LaTeX mode, the header begins with '`\documentstyle`' and ends with '`\begin{document}`'. These are commands that LaTeX requires you to use in any case, so nothing special needs to be done to identify the header.

Entering either kind of TeX mode calls with no arguments the value of the variable `text-mode-hook`, if that value exists and is not `nil`; then it does the same with the variable `TeX-mode-hook`. Finally it does the same with either `plain-TeX-mode-hook` or `LaTeX-mode-hook`.

20.1.3 Outline Mode

Outline mode is a major mode much like Text mode but intended for editing outlines. It allows you to make parts of the text temporarily invisible so that you can see just the overall structure of the outline. Type `M-x outline-mode` to turn on Outline mode in the current buffer.

Entering Outline mode calls with no arguments the value of the variable `text-mode-hook`, if that value exists and is not `nil`; then it does the same with the variable `outline-mode-hook`.

When a line is invisible in outline mode, it does not appear on the screen. The screen appears exactly as if the invisible line were deleted, except that an ellipsis (three periods in a row) appears at the end of the previous visible line (only one ellipsis no matter how many invisible lines follow).

All editing commands treat the text of the invisible line as part of the previous visible line. For example, `C-n` moves onto the next visible line. Killing an entire visible line, including its terminating newline, really kills all the following invisible lines along with it; yanking it all back yanks the invisible lines and they remain invisible.

20.1.3.1 Format of Outlines

Outline mode assumes that the lines in the buffer are of two types: *heading lines* and *body lines*. A heading line represents a topic in the outline. Heading lines start with one or more stars; the number of stars determines the depth of the heading in the outline structure. Thus, a heading line with one star is a major topic; all the heading lines with two stars between it and the next one-star heading are its subtopics; and so on. Any line that is not a heading line is a body line. Body lines belong to the preceding heading line. Here is an example:

```
* Food
```

```
This is the body,
which says something about the topic of food.

** Delicious Food

This is the body of the second-level header.

** Distasteful Food

This could have
a body too, with
several lines.

*** Dormitory Food

* Shelter

A second first-level topic with its header line.
```

A heading line together with all following body lines is called collectively an *entry*. A heading line together with all following deeper heading lines and their body lines is called a *subtree*.

You can customize the criterion for distinguishing heading lines by setting the variable `outline-regexp`. Any line whose beginning has a match for this regexp is considered a heading line. Matches that start within a line (not at the beginning) do not count. The length of the matching text determines the level of the heading; longer matches make a more deeply nested level. Thus, for example, if a text formatter has commands '`@chapter`', '`@section`' and '`@subsection`' to divide the document into chapters and sections, you could make those lines count as heading lines by setting `outline-regexp` to '`"@chap\\|@\\(sub\\)*section"`'. Note the trick: the two words '`chapter`' and '`section`' are equally long, but by defining the regexp to match only '`chap`' we ensure that the length of the text matched on a chapter heading is shorter, so that Outline mode will know that sections are contained in chapters. This works as long as no other command starts with '`@chap`'.

Outline mode makes a line invisible by changing the newline before it into an ASCII Control-M (code 015). Most editing commands that work on lines treat an invisible line as part of the previous line because, strictly speaking, it *is* part of that line, since there is no longer a newline in between. When you save the file in Outline mode, Control-M characters are saved as newlines, so the invisible lines become ordinary lines in the file. But saving does not change the visibility status of a line inside Emacs.

20.1.3.2 Outline Motion Commands

There are some special motion commands in Outline mode that move backward and forward to heading lines.

`C-c C-n` Move point to the next visible heading line (`outline-next-visible-heading`).

`C-c C-p` Move point to the previous visible heading line (`outline-previous-visible-heading`).

`C-c C-f` Move point to the next visible heading line at the same level as the one point is on (`outline-forward-same-level`).

`C-c C-b` Move point to the previous visible heading line at the same level (`outline-backward-same-level`).

`C-c C-u` Move point up to a lower-level (more inclusive) visible heading line (`outline-up-heading`).

`C-c C-n` (`next-visible-heading`) moves down to the next heading line. `C-c C-p` (`previous-visible-heading`) moves similarly backward. Both accept numeric arguments as repeat counts. The names emphasize that invisible headings are skipped, but this is not really a special feature. All editing commands that look for lines ignore the invisible lines automatically.

More advanced motion commands understand the levels of headings. The commands `C-c C-f` (`outline-forward-same-level`) and `C-c C-b` (`outline-backward-same-level`) move from one heading line to another visible heading at the same depth in the outline. `C-c C-u` (`outline-up-heading`) moves backward to another heading that is less deeply nested.

20.1.3.3 Outline Visibility Commands

The other special commands of outline mode are used to make lines visible or invisible. Their names all start with `hide` or `show`. Most of them fall into pairs of opposites. They are not undoable; instead, you can undo right past them. Making lines visible or invisible is simply not recorded by the undo mechanism.

`M-x hide-body`
Make all body lines in the buffer invisible.

`M-x show-all`
Make all lines in the buffer visible.

`C-c C-h` Make everything under this heading invisible, not including this heading itself (`hide-subtree`).

`C-c C-s` Make everything under this heading visible, including body, subheadings, and their bodies (`show-subtree`).

`M-x hide-leaves`
: Make the body of this heading line, and of all its subheadings, invisible.

`M-x show-branches`
: Make all subheadings of this heading line, at all levels, visible.

`C-c C-i` Make immediate subheadings (one level down) of this heading line visible (`show-children`).

`M-x hide-entry`
: Make this heading line's body invisible.

`M-x show-entry`
: Make this heading line's body visible.

Two commands that are exact opposites are `M-x hide-entry` and `M-x show-entry`. They are used with point on a heading line, and apply only to the body lines of that heading. The subtopics and their bodies are not affected.

Two more powerful opposites are `C-c C-h` (`hide-subtree`) and `C-c C-s` (`show-subtree`). Both expect to be used when point is on a heading line, and both apply to all the lines of that heading's *subtree*: its body, all its subheadings, both direct and indirect, and all of their bodies. In other words, the subtree contains everything following this heading line, up to and not including the next heading of the same or higher rank.

Intermediate between a visible subtree and an invisible one is having all the subheadings visible but none of the body. There are two commands for doing this, depending on whether you want to hide the bodies or make the subheadings visible. They are `M-x hide-leaves` and `M-x show-branches`.

A little weaker than `show-branches` is `C-c C-i` (`show-children`). It makes just the direct subheadings visible—those one level down. Deeper subheadings remain invisible, if they were invisible.

Two commands have a blanket effect on the whole file. `M-x hide-body` makes all body lines invisible, so that you see just the outline structure. `M-x show-all` makes all lines visible. These commands can be thought of as a pair of opposites even though `M-x show-all` applies to more than just body lines.

The use of ellipses at the ends of visible lines can be turned off by setting `selective-display-ellipses` to `nil`. Then there is no visible indication of the presence of invisible lines.

20.2 Words

Emacs has commands for moving over or operating on words. By convention, the keys for them are all `Meta-` characters.

`M-f` Move forward over a word (`forward-word`).

`M-b` Move backward over a word (`backward-word`).

`M-d` Kill up to the end of a word (`kill-word`).

`M-DEL` Kill back to the beginning of a word (`backward-kill-word`).

`M-@` Mark the end of the next word (`mark-word`).

`M-t` Transpose two words; drag a word forward or backward across other words (`transpose-words`).

Notice how these keys form a series that parallels the character-based `C-f`, `C-b`, `C-d`, `C-t` and `DEL`. `M-@` is related to `C-@`, which is an alias for `C-SPC`.

The commands `Meta-f` (`forward-word`) and `Meta-b` (`backward-word`) move forward and backward over words. They are thus analogous to `Control-f` and `Control-b`, which move over single characters. Like their `Control-` analogues, `Meta-f` and `Meta-b` move several words if given an argument. `Meta-f` with a negative argument moves backward, and `Meta-b` with a negative argument moves forward. Forward motion stops right after the last letter of the word, while backward motion stops right before the first letter.

`Meta-d` (`kill-word`) kills the word after point. To be precise, it kills everything from point to the place `Meta-f` would move to. Thus, if point is in the middle of a word, `Meta-d` kills just the part after point. If some punctuation comes between point and the next word, it is killed along with the word. (If you wish to kill only the next word but not the punctuation before it, simply do `Meta-f` to get the end, and kill the word backwards with `Meta-DEL`.) `Meta-d` takes arguments just like `Meta-f`.

`Meta-DEL` (`backward-kill-word`) kills the word before point. It kills everything from point back to where `Meta-b` would move to. If point is after the space in '`FOO, BAR`', then '`FOO, `' is killed. (If you wish to kill just '`FOO`', do `Meta-b Meta-d` instead of `Meta-DEL`.)

Meta-t (transpose-words) exchanges the word before or containing point with the following word. The delimiter characters between the words do not move. For example, 'FOO, BAR' transposes into 'BAR, FOO' rather than 'BAR FOO,'. See Section 14.2 [Transpose], page 83, for more on transposition and on arguments to transposition commands.

To operate on the next *n* words with an operation which applies between point and mark, you can either set the mark at point and then move over the words, or you can use the command Meta-@ (mark-word) which does not move point, but sets the mark where Meta-f would move to. It can be given arguments just like Meta-f.

The word commands' understanding of syntax is completely controlled by the syntax table. Any character can, for example, be declared to be a word delimiter. See Section 28.5 [Syntax], page 229.

20.3 Sentences

The Emacs commands for manipulating sentences and paragraphs are mostly on Meta- keys, so as to be like the word-handling commands.

M-a Move back to the beginning of the sentence (backward-sentence).

M-e Move forward to the end of the sentence (forward-sentence).

M-k Kill forward to the end of the sentence (kill-sentence).

C-x DEL Kill back to the beginning of the sentence (backward-kill-sentence).

The commands Meta-a and Meta-e (backward-sentence and forward-sentence) move to the beginning and end of the current sentence, respectively. They were chosen to resemble Control-a and Control-e, which move to the beginning and end of a line. Unlike them, Meta-a and Meta-e if repeated or given numeric arguments move over successive sentences. Emacs assumes that the typist's convention is followed, and thus considers a sentence to end wherever there is a '.', '?' or '!' followed by the end of a line or two spaces, with any number of ')', ']', ''', or '"' characters allowed in between. A sentence also begins or ends wherever a paragraph begins or ends.

Neither M-a nor M-e moves past the newline or spaces beyond the sentence edge at which it is stopping.

Just as C-a and C-e have a kill command, C-k, to go with them, so M-a and M-e have a corresponding kill command M-k (kill-sentence) which kills from point to the end of the sentence.

With minus one as an argument it kills back to the beginning of the sentence. Larger arguments serve as a repeat count.

There is a special command, `C-x DEL` (`backward-kill-sentence`) for killing back to the beginning of a sentence, because this is useful when you change your mind in the middle of composing text.

The variable `sentence-end` controls recognition of the end of a sentence. It is a regexp that matches the last few characters of a sentence, together with the whitespace following the sentence. Its normal value is

```
"[.?!][]\"')]*\\($\\|\t\\|  \\)[ \t\n]*"
```

This example is explained in the section on regexps. See Section 13.5 [Regexps], page 74.

20.4 Paragraphs

The Emacs commands for manipulating paragraphs are also `Meta-` keys.

`M-[`
: Move back to previous paragraph beginning (`backward-paragraph`).

`M-]`
: Move forward to next paragraph end (`forward-paragraph`).

`M-h`
: Put point and mark around this or next paragraph (`mark-paragraph`).

`Meta-[` moves to the beginning of the current or previous paragraph, while `Meta-]` moves to the end of the current or next paragraph. Blank lines and text formatter command lines separate paragraphs and are not part of any paragraph. Also, an indented line starts a new paragraph.

In major modes for programs (as opposed to Text mode), paragraphs begin and end only at blank lines. This makes the paragraph commands continue to be useful even though there are no paragraphs per se.

When there is a fill prefix, then paragraphs are delimited by all lines which don't start with the fill prefix. See Section 20.6 [Filling], page 134.

When you wish to operate on a paragraph, you can use the command `Meta-h` (`mark-paragraph`) to set the region around it. This command puts point at the beginning and mark at the end of the paragraph point was in. If point is between paragraphs (in a run of blank lines, or at a boundary), the paragraph following point is surrounded by point and mark. If there are blank lines preceding the first line of the paragraph, one of these blank lines is included in the region. Thus, for example, `M-h C-w` kills the paragraph around or after point.

The precise definition of a paragraph boundary is controlled by the variables `paragraph-separate` and `paragraph-start`. The value of `paragraph-start` is a regexp that should match any line that either starts or separates paragraphs. The value of `paragraph-separate` is another regexp that should match only lines that separate paragraphs without being part of any paragraph. Lines that start a new paragraph and are contained in it must match both regexps. For example, normally `paragraph-start` is `"^[ \t\n\f]"` and `paragraph-separate` is `"^[ \t\f]*$"`.

Normally it is desirable for page boundaries to separate paragraphs. The default values of these variables recognize the usual separator for pages.

20.5 Pages

Files are often thought of as divided into *pages* by the *formfeed* character (ASCII Control-L, octal code 014). For example, if a file is printed on a line printer, each page of the file, in this sense, will start on a new page of paper. Emacs treats a page-separator character just like any other character. It can be inserted with `C-q C-l`, or deleted with `DEL`. Thus, you are free to paginate your file or not. However, since pages are often meaningful divisions of the file, commands are provided to move over them and operate on them.

`C-x [` Move point to previous page boundary (`backward-page`).

`C-x ]` Move point to next page boundary (`forward-page`).

`C-x C-p` Put point and mark around this page (or another page) (`mark-page`).

`C-x l` Count the lines in this page (`count-lines-page`).

The `C-x [` (`backward-page`) command moves point to immediately after the previous page delimiter. If point is already right after a page delimiter, it skips that one and stops at the previous one. A numeric argument serves as a repeat count. The `C-x ]` (`forward-page`) command moves forward past the next page delimiter.

The `C-x C-p` command (`mark-page`) puts point at the beginning of the current page and the mark at the end. The page delimiter at the end is included (the mark follows it). The page delimiter at the front is excluded (point follows it). This command can be followed by `C-w` to kill a page which is to be moved elsewhere. If it is inserted after a page delimiter, at a place where `C-x ]` or `C-x [` would take you, then the page will be properly delimited before and after once again.

A numeric argument to `C-x C-p` is used to specify which page to go to, relative to the current one. Zero means the current page. One means the next page, and −1 means the previous one.

The `C-x l` command (`count-lines-page`) is good for deciding where to break a page in two. It prints in the echo area the total number of lines in the current page, and then divides it up into those preceding the current line and those following, as in

```
Page has 96 (72+25) lines
```

Notice that the sum is off by one; this is correct if point is not at the beginning of a line.

The variable `page-delimiter` should have as its value a regexp that matches the beginning of a line that separates pages. This is what defines where pages begin. The normal value of this variable is `"^\f"`, which matches a formfeed character at the beginning of a line.

20.6 Filling Text

With Auto Fill mode, text can be *filled* (broken up into lines that fit in a specified width) as you insert it. If you alter existing text it may no longer be properly filled; then explicit commands for filling can be used.

20.6.1 Auto Fill Mode

Auto Fill mode is a minor mode in which lines are broken automatically when they become too wide. Breaking happens only when you type a `SPC` or `RET`.

`M-x auto-fill-mode`
: Enable or disable Auto Fill mode.

`SPC`
`RET`
: In Auto Fill mode, break lines when appropriate.

`M-x auto-fill-mode` turns Auto Fill mode on if it was off, or off if it was on. With a positive numeric argument it always turns Auto Fill mode on, and with a negative argument always turns it off. You can see when Auto Fill mode is in effect by the presence of the word 'Fill' in the mode line, inside the parentheses. Auto Fill mode is a minor mode, turned on or off for each buffer individually. See Section 28.1 [Minor Modes], page 217.

In Auto Fill mode, lines are broken automatically at spaces when they get longer than the desired width. Line breaking and rearrangement takes place only when you type SPC or RET. If you wish to insert a space or newline without permitting line-breaking, type C-q SPC or C-q LFD (recall that a newline is really a linefeed). Also, C-o inserts a newline without line breaking.

Auto Fill mode works well with Lisp mode, because when it makes a new line in Lisp mode it indents that line with TAB. If a line ending in a comment gets too long, the text of the comment is split into two comment lines. Optionally new comment delimiters are inserted at the end of the first line and the beginning of the second so that each line is a separate comment; the variable comment-multi-line controls the choice (see Section 21.6.2 [Comments], page 152).

Auto Fill mode does not refill entire paragraphs. It can break lines but cannot merge lines. So editing in the middle of a paragraph can result in a paragraph that is not correctly filled. The easiest way to make the paragraph properly filled again is usually with the explicit fill commands.

Many users like Auto Fill mode and want to use it in all text files. The section on init files says how to arrange this permanently for yourself. See Section 28.6 [Init File], page 232.

20.6.2 Explicit Fill Commands

M-q
Fill current paragraph (fill-paragraph).

M-g
Fill each paragraph in the region (fill-region).

C-x f
Set the fill column (set-fill-column).

M-x fill-region-as-paragraph.
Fill the region, considering it as one paragraph.

M-s
Center a line.

To refill a paragraph, use the command Meta-q (fill-paragraph). It causes the paragraph that point is inside, or the one after point if point is between paragraphs, to be refilled. All the line-breaks are removed, and then new ones are inserted where necessary. M-q can be undone with C-_. See Chapter 5 [Undo], page 33.

To refill many paragraphs, use M-g (fill-region), which divides the region into paragraphs and fills each of them.

`Meta-q` and `Meta-g` use the same criteria as `Meta-h` for finding paragraph boundaries (see Section 20.4 [Paragraphs], page 132). For more control, you can use `M-x fill-region-as-paragraph`, which refills everything between point and mark. This command recognizes only blank lines as paragraph separators.

A numeric argument to `M-g` or `M-q` causes it to *justify* the text as well as filling it. This means that extra spaces are inserted to make the right margin line up exactly at the fill column. To remove the extra spaces, use `M-q` or `M-g` with no argument.

The command `Meta-s` (`center-line`) centers the current line within the current fill column. With an argument, it centers several lines individually and moves past them.

The maximum line width for filling is in the variable `fill-column`. Altering the value of `fill-column` makes it local to the current buffer; until that time, the default value is in effect. The default is initially 70. See Section 28.2.3 [Locals], page 220.

The easiest way to set `fill-column` is to use the command `C-x f` (`set-fill-column`). With no argument, it sets `fill-column` to the current horizontal position of point. With a numeric argument, it uses that as the new fill column.

20.6.3 The Fill Prefix

To fill a paragraph in which each line starts with a special marker (which might be a few spaces, giving an indented paragraph), use the *fill prefix* feature. The fill prefix is a string which Emacs expects every line to start with, and which is not included in filling.

`C-x .`
: Set the fill prefix (`set-fill-prefix`).

`M-q`
: Fill a paragraph using current fill prefix (`fill-paragraph`).

`M-x fill-individual-paragraphs`
: Fill the region, considering each change of indentation as starting a new paragraph.

To specify a fill prefix, move to a line that starts with the desired prefix, put point at the end of the prefix, and give the command `C-x .` (`set-fill-prefix`). That's a period after the `C-x`. To turn off the fill prefix, specify an empty prefix: type `C-x .` with point at the beginning of a line.

When a fill prefix is in effect, the fill commands remove the fill prefix from each line before filling and insert it on each line after filling. The fill prefix is also inserted on new lines made automatically by Auto Fill mode. Lines that do not start with the fill prefix are considered to start paragraphs, both in `M-q` and the paragraph commands; this is just right if you are using paragraphs with hanging indentation (every line indented except the first one). Lines which are blank or indented once the prefix is removed also separate or start paragraphs; this is what you want if you are writing multi-paragraph comments with a comment delimiter on each line.

The fill prefix is stored in the variable `fill-prefix`. Its value is a string, or `nil` when there is no fill prefix. This is a per-buffer variable; altering the variable affects only the current buffer, but there is a default value which you can change as well. See Section 28.2.3 [Locals], page 220.

Another way to use fill prefixes is through `M-x fill-individual-paragraphs`. This function divides the region into groups of consecutive lines with the same amount and kind of indentation and fills each group as a paragraph using its indentation as a fill prefix.

20.7 Case Conversion Commands

Emacs has commands for converting either a single word or any arbitrary range of text to upper case or to lower case.

`M-l` Convert following word to lower case (`downcase-word`).

`M-u` Convert following word to upper case (`upcase-word`).

`M-c` Capitalize the following word (`capitalize-word`).

`C-x C-l` Convert region to lower case (`downcase-region`).

`C-x C-u` Convert region to upper case (`upcase-region`).

The word conversion commands are the most useful. `Meta-l` (`downcase-word`) converts the word after point to lower case, moving past it. Thus, repeating `Meta-l` converts successive words. `Meta-u` (`upcase-word`) converts to all capitals instead, while `Meta-c` (`capitalize-word`) puts the first letter of the word into upper case and the rest into lower case. All these commands convert several words at once if given an argument. They are especially convenient for converting a large amount of text from all upper case to mixed case, because you can move through the text using `M-l`, `M-u` or `M-c` on each word as appropriate, occasionally using `M-f` instead to skip a word.

When given a negative argument, the word case conversion commands apply to the appropriate number of words before point, but do not move point. This is convenient when you have just typed a word in the wrong case: you can give the case conversion command and continue typing.

If a word case conversion command is given in the middle of a word, it applies only to the part of the word which follows point. This is just like what `Meta-d` (`kill-word`) does. With a negative argument, case conversion applies only to the part of the word before point.

The other case conversion commands are `C-x C-u` (`upcase-region`) and `C-x C-l` (`downcase-region`), which convert everything between point and mark to the specified case. Point and mark do not move.

21 Editing Programs

Emacs has many commands designed to understand the syntax of programming languages such as Lisp and C. These commands can

- Move over or kill balanced expressions or *sexps* (see Section 21.2 [Lists], page 140).
- Move over or mark top-level balanced expressions (*defuns*, in Lisp; functions, in C).
- Show how parentheses balance (see Section 21.5 [Matching], page 149).
- Insert, kill or align comments (see Section 21.6.2 [Comments], page 152).
- Follow the usual indentation conventions of the language (see Section 21.4 [Grinding], page 143).

The commands for words, sentences and paragraphs are very useful in editing code even though their canonical application is for editing human language text. Most symbols contain words (see Section 20.2 [Words], page 130); sentences can be found in strings and comments (see Section 20.3 [Sentences], page 131). Paragraphs per se are not present in code, but the paragraph commands are useful anyway, because Lisp mode and C mode define paragraphs to begin and end at blank lines (see Section 20.4 [Paragraphs], page 132). Judicious use of blank lines to make the program clearer will also provide interesting chunks of text for the paragraph commands to work on.

The selective display feature is useful for looking at the overall structure of a function (see Section 12.3 [Selective Display], page 67). This feature causes only the lines that are indented less than a specified amount to appear on the screen.

21.1 Major Modes for Programming Languages

Emacs also has major modes for the programming languages Lisp, Scheme (a variant of Lisp), C, Fortran and Muddle. Ideally, a major mode should be implemented for each programming language that you might want to edit with Emacs; but often the mode for one language can serve for other syntactically similar languages. The language modes that exist are those that someone decided to take the trouble to write.

There are several forms of Lisp mode, which differ in the way they interface to Lisp execution. See Section 22.2 [Lisp Modes], page 169.

Each of the programming language modes defines the `TAB` key to run an indentation function that knows the indentation conventions of that language and updates the current line's indentation accordingly. For example, in C mode `TAB` is bound to `c-indent-line`. `LFD` is normally defined to do `RET` followed by `TAB`; thus, it too indents in a mode-specific fashion.

In most programming languages, indentation is likely to vary from line to line. So the major modes for those languages rebind `DEL` to treat a tab as if it were the equivalent number of spaces (using the command `backward-delete-char-untabify`). This makes it possible to rub out indentation one column at a time without worrying whether it is made up of spaces or tabs. Use `C-b C-d` to delete a tab character before point, in these modes.

Programming language modes define paragraphs to be separated only by blank lines, so that the paragraph commands remain useful. Auto Fill mode, if enabled in a programming language major mode, indents the new lines which it creates.

Turning on a major mode calls a user-supplied function called the *mode hook*, which is the value of a Lisp variable. For example, turning on C mode calls the value of the variable `c-mode-hook` if that value exists and is non-`nil`. Mode hook variables for other programming language modes include `lisp-mode-hook`, `emacs-lisp-mode-hook`, `lisp-interaction-mode-hook`, `scheme-mode-hook` and `muddle-mode-hook`. The mode hook function receives no arguments.

21.2 Lists and Sexps

By convention, Emacs keys for dealing with balanced expressions are usually `Control-Meta-` characters. They tend to be analogous in function to their `Control-` and `Meta-` equivalents. These commands are usually thought of as pertaining to expressions in programming languages, but can be useful with any language in which some sort of parentheses exist (including English).

These commands fall into two classes. Some deal only with *lists* (parenthetical groupings). They see nothing except parentheses, brackets, braces (whichever ones must balance in the language you are working with), and escape characters that might be used to quote those.

The other commands deal with expressions or *sexps*. The word 'sexp' is derived from *s-expression*, the ancient term for an expression in Lisp. But in Emacs, the notion of 'sexp' is not limited to Lisp. It refers to an expression in whatever language your program is written in. Each programming language has its own major mode, which customizes the syntax tables so that expressions in that language count as sexps.

Sexps typically include symbols, numbers, and string constants, as well as anything contained in parentheses, brackets or braces.

In languages that use prefix and infix operators, such as C, it is not possible for all expressions to be sexps. For example, C mode does not recognize '`foo + bar`' as a sexp, even though it *is* a C expression; it recognizes '`foo`' as one sexp and '`bar`' as another, with the '`+`' as punctuation between them. This is a fundamental ambiguity: both '`foo + bar`' and '`foo`' are legitimate choices for the sexp to move over if point is at the '`f`'. Note that '`(foo + bar)`' is a sexp in C mode.

Some languages have obscure forms of syntax for expressions that nobody has bothered to make Emacs understand properly.

`C-M-f` Move forward over a sexp (`forward-sexp`).

`C-M-b` Move backward over a sexp (`backward-sexp`).

`C-M-k` Kill sexp forward (`kill-sexp`).

`C-M-u` Move up and backward in list structure (`backward-up-list`).

`C-M-d` Move down and forward in list structure (`down-list`).

`C-M-n` Move forward over a list (`forward-list`).

`C-M-p` Move backward over a list (`backward-list`).

`C-M-t` Transpose expressions (`transpose-sexps`).

`C-M-@` Put mark after following expression (`mark-sexp`).

To move forward over a sexp, use `C-M-f` (`forward-sexp`). If the first significant character after point is an opening delimiter ('`(`' in Lisp; '`(`', '`[`' or '`{`' in C), `C-M-f` moves past the matching closing delimiter. If the character begins a symbol, string, or number, `C-M-f` moves over that. If the character after point is a closing delimiter, `C-M-f` just moves past it. (This last is not really moving across a sexp; it is an exception which is included in the definition of `C-M-f` because it is as useful a behavior as anyone can think of for that situation.)

The command `C-M-b` (`backward-sexp`) moves backward over a sexp. The detailed rules are like those above for `C-M-f`, but with directions reversed. If there are any prefix characters (singlequote, backquote and comma, in Lisp) preceding the sexp, `C-M-b` moves back over them as well.

`C-M-f` or `C-M-b` with an argument repeats that operation the specified number of times; with a negative argument, it moves in the opposite direction.

The sexp commands move across comments as if they were whitespace, in languages such as C where the comment-terminator can be recognized. In Lisp, and other languages where comments run until the end of a line, it is very difficult to ignore comments when parsing backwards; therefore, in such languages the sexp commands treat the text of comments as if it were code.

Killing a sexp at a time can be done with `C-M-k` (`kill-sexp`). `C-M-k` kills the characters that `C-M-f` would move over.

The *list commands* move over lists like the sexp commands but skip blithely over any number of other kinds of sexps (symbols, strings, etc). They are `C-M-n` (`forward-list`) and `C-M-p` (`backward-list`). The main reason they are useful is that they usually ignore comments (since the comments usually do not contain any lists).

`C-M-n` and `C-M-p` stay at the same level in parentheses, when that's possible. To move *up* one (or *n*) levels, use `C-M-u` (`backward-up-list`). `C-M-u` moves backward up past one unmatched opening delimiter. A positive argument serves as a repeat count; a negative argument reverses direction of motion and also requests repetition, so it moves forward and up one or more levels.

To move *down* in list structure, use `C-M-d` (`down-list`). In Lisp mode, where '`(`' is the only opening delimiter, this is nearly the same as searching for a '`(`'. An argument specifies the number of levels of parentheses to go down.

A somewhat random-sounding command which is nevertheless easy to use is `C-M-t` (`transpose-sexps`), which drags the previous sexp across the next one. An argument serves as a repeat count, and a negative argument drags backwards (thus canceling out the effect of `C-M-t` with a positive argument). An argument of zero, rather than doing nothing, transposes the sexps ending after point and the mark.

To make the region be the next sexp in the buffer, use `C-M-@` (`mark-sexp`) which sets mark at the same place that `C-M-f` would move to. `C-M-@` takes arguments like `C-M-f`. In particular, a negative argument is useful for putting the mark at the beginning of the previous sexp.

The list and sexp commands' understanding of syntax is completely controlled by the syntax table. Any character can, for example, be declared to be an opening delimiter and act like an open parenthesis. See Section 28.5 [Syntax], page 229.

21.3 Defuns

In Emacs, a parenthetical grouping at the top level in the buffer is called a *defun*. The name derives from the fact that most top-level lists in a Lisp file are instances of the special form `defun`, but any top-level parenthetical grouping counts as a defun in Emacs parlance regardless of what its contents are, and regardless of the programming language in use. For example, in C, the body of a function definition is a defun.

`C-M-a` Move to beginning of current or preceding defun (`beginning-of-defun`).

`C-M-e` Move to end of current or following defun (`end-of-defun`).

`C-M-h` Put region around whole current or following defun (`mark-defun`).

The commands to move to the beginning and end of the current defun are `C-M-a` (`beginning-of-defun`) and `C-M-e` (`end-of-defun`).

If you wish to operate on the current defun, use `C-M-h` (`mark-defun`) which puts point at the beginning and mark at the end of the current or next defun. For example, this is the easiest way to get ready to move the defun to a different place in the text. In C mode, `C-M-h` runs the function `mark-c-function`, which is almost the same as `mark-defun`; the difference is that it backs up over the argument declarations, function name and returned data type so that the entire C function is inside the region.

Emacs assumes that any open-parenthesis found in the leftmost column is the start of a defun. Therefore, **never put an open-parenthesis at the left margin in a Lisp file unless it is the start of a top level list. Never put an open-brace or other opening delimiter at the beginning of a line of C code unless it starts the body of a function.** The most likely problem case is when you want an opening delimiter at the start of a line inside a string. To avoid trouble, put an escape character ('`\`', in C and Emacs Lisp, '`/`' in some other Lisp dialects) before the opening delimiter. It will not affect the contents of the string.

In the remotest past, the original Emacs found defuns by moving upward a level of parentheses until there were no more levels to go up. This always required scanning all the way back to the beginning of the buffer, even for a small function. To speed up the operation, Emacs was changed to assume that any '`(`' (or other character assigned the syntactic class of opening-delimiter) at the left margin is the start of a defun. This heuristic was nearly always right and avoided the costly scan; however, it mandated the convention described above.

21.4 Indentation for Programs

The best way to keep a program properly indented ("ground") is to use Emacs to re-indent it as you change it. Emacs has commands to indent properly either a single line, a specified number of lines, or all of the lines inside a single parenthetical grouping.

21.4.1 Basic Program Indentation Commands

`TAB` Adjust indentation of current line.

`LFD` Equivalent to `RET` followed by `TAB` (`newline-and-indent`).

The basic indentation command is `TAB`, which gives the current line the correct indentation as determined from the previous lines. The function that `TAB` runs depends on the major mode; it is `lisp-indent-line` in Lisp mode, `c-indent-line` in C mode, etc. These functions understand different syntaxes for different languages, but they all do about the same thing. `TAB` in any programming language major mode inserts or deletes whitespace at the beginning of the current line, independent of where point is in the line. If point is inside the whitespace at the beginning of the line, `TAB` leaves it at the end of that whitespace; otherwise, `TAB` leaves point fixed with respect to the characters around it.

Use `C-q TAB` to insert a tab at point.

When entering a large amount of new code, use `LFD` (`newline-and-indent`), which is equivalent to a `RET` followed by a `TAB`. `LFD` creates a blank line, and then gives it the appropriate indentation.

`TAB` indents the second and following lines of the body of an parenthetical grouping each under the preceding one; therefore, if you alter one line's indentation to be nonstandard, the lines below will tend to follow it. This is the right behavior in cases where the standard result of `TAB` is unaesthetic.

Remember that an open-parenthesis, open-brace or other opening delimiter at the left margin is assumed by Emacs (including the indentation routines) to be the start of a function. Therefore, you must never have an opening delimiter in column zero that is not the beginning of a function, not even inside a string. This restriction is vital for making the indentation commands fast; you must simply accept it. See Section 21.3 [Defuns], page 142, for more information on this.

21.4.2 Indenting Several Lines

When you wish to re-indent several lines of code which have been altered or moved to a different level in the list structure, you have several commands available.

`C-M-q` Re-indent all the lines within one list (`indent-sexp`).

`C-u TAB` Shift an entire list rigidly sideways so that its first line is properly indented.

`C-M-\` Re-indent all lines in the region (`indent-region`).

You can re-indent the contents of a single list by positioning point before the beginning of it and typing `C-M-q` (`indent-sexp` in Lisp mode, `indent-c-exp` in C mode; also bound to other suitable functions in other modes). The indentation of the line the sexp starts on is not changed; therefore, only the relative indentation within the list, and not its position, is changed. To correct the position as well, type a `TAB` before the `C-M-q`.

If the relative indentation within a list is correct but the indentation of its beginning is not, go to the line the list begins on and type `C-u TAB`. When `TAB` is given a numeric argument, it moves all the lines in the grouping starting on the current line sideways the same amount that the current line moves. It is clever, though, and does not move lines that start inside strings, or C preprocessor lines when in C mode.

Another way to specify the range to be re-indented is with point and mark. The command `C-M-\` (`indent-region`) applies `TAB` to every line whose first character is between point and mark.

21.4.3 Customizing Lisp Indentation

The indentation pattern for a Lisp expression can depend on the function called by the expression. For each Lisp function, you can choose among several predefined patterns of indentation, or define an arbitrary one with a Lisp program.

The standard pattern of indentation is as follows: the second line of the expression is indented under the first argument, if that is on the same line as the beginning of the expression; otherwise, the second line is indented underneath the function name. Each following line is indented under the previous line whose nesting depth is the same.

If the variable `lisp-indent-offset` is non-`nil`, it overrides the usual indentation pattern for the second line of an expression, so that such lines are always indented `lisp-indent-offset` more columns than the containing list.

The standard pattern is overridded for certain functions. Functions whose names start with `def` always indent the second line by `lisp-body-indention` extra columns beyond the open-parenthesis starting the expression.

The standard pattern can be overridden in various ways for individual functions, according to the `lisp-indent-hook` property of the function name. There are four possibilities for this property:

`nil` This is the same as no property; the standard indentation pattern is used.

`defun` The pattern used for function names that start with `def` is used for this function also.

a number, *number*

The first *number* arguments of the function are *distinguished* arguments; the rest are considered the *body* of the expression. A line in the expression is indented according to whether the first argument on it is distinguished or not. If the argument is part of the body, the line is indented `lisp-body-indent` more columns than the open-parenthesis starting the containing expression. If the argument is distinguished and is either the first or second argument, it is indented *twice* that many extra columns. If the argument is distinguished and not the first or second argument, the standard pattern is followed for that line.

a symbol, *symbol*

symbol should be a function name; that function is called to calculate the indentation of a line within this expression. The function receives two arguments:

state The value returned by `parse-partial-sexp` (a Lisp primitive for indentation and nesting computation) when it parses up to the beginning of this line.

pos The position at which the line being indented begins.

It should return either a number, which is the number of columns of indentation for that line, or a list whose car is such a number. The difference between returning a number and returning a list is that a number says that all following lines at the same nesting level should be indented just like this one; a list says that following lines might call for different indentations. This makes a difference when the indentation is being computed by `C-M-q`; if the value is a number, `C-M-q` need not recalculate indentation for the following lines until the end of the list.

21.4.4 Customizing C Indentation

Two variables control which commands perform C indentation and when.

If `c-auto-newline` is non-`nil`, newlines are inserted both before and after braces that you insert, and after colons and semicolons. Correct C indentation is done on all the lines that are made this way.

If `c-tab-always-indent` is non-`nil`, the `TAB` command in C mode does indentation only if point is at the left margin or within the line's indentation. If there is non-whitespace to the left of point, then `TAB` just inserts a tab character in the buffer. Normally, this variable is `nil`, and `TAB` always reindents the current line.

C does not have anything analogous to particular function names for which special forms of indentation are desirable. However, it has a different need for customization facilities: many different styles of C indentation are in common use.

There are six variables you can set to control the style that Emacs C mode will use.

`c-indent-level`
: Indentation of C statements within surrounding block. The surrounding block's indentation is the indentation of the line on which the open-brace appears.

`c-continued-statement-offset`
: Extra indentation given to a substatement, such as the then-clause of an if or body of a while.

`c-brace-offset`
: Extra indentation for line if it starts with an open brace.

`c-brace-imaginary-offset`
: An open brace following other text is treated as if it were this far to the right of the start of its line.

`c-argdecl-indent`
: Indentation level of declarations of C function arguments.

`c-label-offset`
: Extra indentation for line that is a label, or case or default.

The variable `c-indent-level` controls the indentation for C statements with respect to the surrounding block. In the example

```
    {
      foo ();
```

the difference in indentation between the lines is `c-indent-level`. Its standard value is 2.

If the open-brace beginning the compound statement is not at the beginning of its line, the `c-indent-level` is added to the indentation of the line, not the column of the open-brace. For example,

```
if (losing) {
  do_this ();
```

One popular indentation style is that which results from setting `c-indent-level` to 8 and putting open-braces at the end of a line in this way. I prefer to put the open-brace on a separate line.

In fact, the value of the variable `c-brace-imaginary-offset` is also added to the indentation of such a statement. Normally this variable is zero. Think of this variable as the imaginary position of the open brace, relative to the first nonblank character on the line. By setting this variable to 4 and `c-indent-level` to 0, you can get this style:

```
if (x == y) {
    do_it ();
    }
```

When `c-indent-level` is zero, the statements inside most braces will line up right under the open brace. But there is an exception made for braces in column zero, such as surrounding a function's body. The statements just inside it do not go at column zero. Instead, `c-brace-offset` and `c-continued-statement-offset` (see below) are added to produce a typical offset between brace levels, and the statements are indented that far.

`c-continued-statement-offset` controls the extra indentation for a line that starts within a statement (but not within parentheses or brackets). These lines are usually statements that are within other statements, such as the then-clauses of `if` statements and the bodies of `while` statements. This parameter is the difference in indentation between the two lines in

```
if (x == y)
  do_it ();
```

Its standard value is 2. Some popular indentation styles correspond to a value of zero for `c-continued-statement-offset`.

`c-brace-offset` is the extra indentation given to a line that starts with an open-brace. Its standard value is zero; compare

```
if (x == y)
  {
```

with

```
if (x == y)
  do_it ();
```

if `c-brace-offset` were set to 4, the first example would become

```
if (x == y)
      {
```

`c-argdecl-indent` controls the indentation of declarations of the arguments of a C function. It is absolute: argument declarations receive exactly `c-argdecl-indent` spaces. The standard value is 5, resulting in code like this:

```
char *
index (string, char)
     char *string;
     int char;
```

`c-label-offset` is the extra indentation given to a line that contains a label, a case statement, or a `default:` statement. Its standard value is −2, resulting in code like this

```
switch (c)
  {
  case 'x':
```

If `c-label-offset` were zero, the same code would be indented as

```
switch (c)
  {
    case 'x':
```

This example assumes that the other variables above also have their standard values.

I strongly recommend that you try out the indentation style produced by the standard settings of these variables, together with putting open braces on separate lines. You can see how it looks in all the C source files of GNU Emacs.

21.5 Automatic Display Of Matching Parentheses

The Emacs parenthesis-matching feature is designed to show automatically how parentheses match in the text. Whenever a self-inserting character that is a closing delimiter is typed, the cursor moves momentarily to the location of the matching opening delimiter, provided that is on the screen. If it is not on the screen, some text starting with that opening delimiter is displayed in the echo area. Either way, you can tell what grouping is being closed off.

In Lisp, automatic matching applies only to parentheses. In C, it applies to braces and brackets too. Emacs knows which characters to regard as matching delimiters based on the syntax table, which is set by the major mode. See Section 28.5 [Syntax], page 229.

If the opening delimiter and closing delimiter are mismatched—such as in '`[x)`'—a warning message is displayed in the echo area. The correct matches are specified in the syntax table.

Two variables control parenthesis match display. `blink-matching-paren` turns the feature on or off; `nil` turns it off, but the default is `t` to turn match display on. `blink-matching-paren-distance` specifies how many characters back to search to find the matching opening delimiter. If the match is not found in that far, scanning stops, and nothing is displayed. This is to prevent scanning for the matching delimiter from wasting lots of time when there is no match. The default is 4000.

21.6 Manipulating Comments

The comment commands insert, kill and align comments.

`M-;` Insert or align comment (`indent-for-comment`).

`C-x ;` Set comment column (`set-comment-column`).

`C-u - C-x ;`
Kill comment on current line (`kill-comment`).

`M-LFD` Like `RET` followed by inserting and aligning a comment (`indent-new-comment-line`).

The command that creates a comment is `Meta-;` (`indent-for-comment`). If there is no comment already on the line, a new comment is created, aligned at a specific column called the *comment column*. The comment is created by inserting the string Emacs thinks comments should start with (the value of `comment-start`; see below). Point is left after that string. If the text of the line extends past the comment column, then the indentation is done to a suitable boundary (usually, at least one space is inserted). If the major mode has specified a string to terminate comments, that is inserted after point, to keep the syntax valid.

`Meta-;` can also be used to align an existing comment. If a line already contains the string that starts comments, then `M-;` just moves point after it and re-indents it to the conventional place. Exception: comments starting in column 0 are not moved.

Some major modes have special rules for indenting certain kinds of comments in certain contexts.

For example, in Lisp code, comments which start with two semicolons are indented as if they were lines of code, instead of at the comment column. Comments which start with three semicolons are supposed to start at the left margin. Emacs understands these conventions by indenting a double-semicolon comment using TAB, and by not changing the indentation of a triple-semicolon comment at all.

```
;; This function is just an example
;;; Here either two or three semicolons are appropriate.
(defun foo (x)
;;; And now, the first part of the function:
  ;; The following line adds one.
  (1+ x))           ; This line adds one.
```

In C code, a comment preceded on its line by nothing but whitespace is indented like a line of code.

Even when an existing comment is properly aligned, M-; is still useful for moving directly to the start of the comment.

C-u - C-x ; (kill-comment) kills the comment on the current line, if there is one. The indentation before the start of the comment is killed as well. If there does not appear to be a comment in the line, nothing is done. To reinsert the comment on another line, move to the end of that line, do C-y, and then do M-; to realign it. Note that C-u - C-x ; is not a distinct key; it is C-x ; (set-comment-column) with a negative argument. That command is programmed so that when it receives a negative argument it calls kill-comment. However, kill-comment is a valid command which you could bind directly to a key if you wanted to.

21.6.1 Multiple Lines of Comments

If you are typing a comment and find that you wish to continue it on another line, you can use the command Meta-LFD (indent-new-comment-line), which terminates the comment you are typing, creates a new blank line afterward, and begins a new comment indented under the old one. When Auto Fill mode is on, going past the fill column while typing a comment causes the comment to be continued in just this fashion. If point is not at the end of the line when M-LFD is typed, the text on the rest of the line becomes part of the new comment line.

21.6.2 Options Controlling Comments

The comment column is stored in the variable `comment-column`. You can set it to a number explicitly. Alternatively, the command `C-x ;` (`set-comment-column`) sets the comment column to the column point is at. `C-u C-x ;` sets the comment column to match the last comment before point in the buffer, and then does a `Meta-;` to align the current line's comment under the previous one. Note that `C-u - C-x ;` runs the function `kill-comment` as described above.

`comment-column` is a per-buffer variable; altering the variable affects only the current buffer, but there is a default value which you can change as well. See Section 28.2.3 [Locals], page 220. Many major modes initialize this variable for the current buffer.

The comment commands recognize comments based on the regular expression that is the value of the variable `comment-start-skip`. This regexp should not match the null string. It may match more than the comment starting delimiter in the strictest sense of the word; for example, in C mode the value of the variable is `"/\\*+ *"`, which matches extra stars and spaces after the '`/*`' itself. (Note that '`\\`' is needed in Lisp syntax to include a '`\`' in the string, which is needed to deny the first star its special meaning in regexp syntax. See Section 13.5 [Regexps], page 74.)

When a comment command makes a new comment, it inserts the value of `comment-start` to begin it. The value of `comment-end` is inserted after point, so that it will follow the text that you will insert into the comment. In C mode, `comment-start` has the value `"/* "` and `comment-end` has the value `" */"`.

`comment-multi-line` controls how `M-LFD` (`indent-new-comment-line`) behaves when used inside a comment. If `comment-multi-line` is `nil`, as it normally is, then the comment on the starting line is terminated and a new comment is started on the new following line. If `comment-multi-line` is not `nil`, then the new following line is set up as part of the same comment that was found on the starting line. This is done by not inserting a terminator on the old line, and not inserting a starter on the new line. In languages where multi-line comments work, the choice of value for this variable is a matter of taste.

The variable `comment-indent-hook` should contain a function that will be called to compute the indentation for a newly inserted comment or for aligning an existing comment. It is set differently by various major modes. The function is called with no arguments, but with point at the beginning of the comment, or at the end of a line if a new comment is to be inserted. It should return the column in which the comment ought to start. For example, in Lisp mode, the indent hook function bases its decision on how many semicolons begin an existing comment, and on the code in the preceding lines.

21.7 Editing Without Unbalanced Parentheses

`M-(` Put parentheses around next sexp(s) (`insert-parentheses`).

`M-)` Move past next close parenthesis and re-indent (`move-over-close-and-reindent`).

The commands `M-(` (`insert-parentheses`) and `M-)` (`move-over-close-and-reindent`) are designed to facilitate a style of editing which keeps parentheses balanced at all times. `M-(` inserts a pair of parentheses, either together as in '`()`', or, if given an argument, around the next several sexps, and leaves point after the open parenthesis. Instead of typing `( F O O )`, you can type `M-( F O O`, which has the same effect except for leaving the cursor before the close parenthesis. Then you would type `M-)`, which moves past the close parenthesis, deleting any indentation preceding it (in this example there is none), and indenting with `LFD` after it.

21.8 Completion for Lisp Symbols

Usually completion happens in the minibuffer. But one kind of completion is available in all buffers: completion for Lisp symbol names.

The command `M-TAB` (`lisp-complete-symbol`) takes the partial Lisp symbol before point to be an abbreviation, and compares it against all nontrivial Lisp symbols currently known to Emacs. Any additional characters that they all have in common are inserted at point. Nontrivial symbols are those that have function definitions, values or properties.

If there is an open-parenthesis immediately before the beginning of the partial symbol, only symbols with function definitions are considered as completions.

If the partial name in the buffer has more than one possible completion and they have no additional characters in common, a list of all possible completions is displayed in another window.

21.9 Documentation Commands

As you edit Lisp code to be run in Emacs, the commands `C-h f` (`describe-function`) and `C-h v` (`describe-variable`) can be used to print documentation of functions and variables that you want to call. These commands use the minibuffer to read the name of a function or variable to document, and display the documentation in a window.

For extra convenience, these commands provide default arguments based on the code in the neighborhood of point. `C-h f` sets the default to the function called in the innermost list containing point. `C-h v` uses the symbol name around or adjacent to point as its default.

Documentation on Unix commands, system calls and libraries can be obtained with the `M-x manual-entry` command. This reads a topic as an argument, and displays the text on that topic from the Unix manual. `manual-entry` always searches all 8 sections of the manual, and concatenates all the entries that are found. For example, the topic '`termcap`' finds the description of the termcap library from section 3, followed by the description of the termcap data base from section 5.

21.10 Change Logs

The Emacs command `M-x add-change-log-entry` helps you keep a record of when and why you have changed a program. It assumes that you have a file in which you write a chronological sequence of entries describing individual changes. The default is to store the change entries in a file called '`ChangeLog`' in the same directory as the file you are editing. The same '`ChangeLog`' file therefore records changes for all the files in the directory.

A change log entry starts with a header line that contains your name and the current date. Aside from these header lines, every line in the change log starts with a tab. One entry can describe several changes; each change starts with a line starting with a tab and a star. `M-x add-change-log-entry` visits the change log file and creates a new entry unless the most recent entry is for today's date and your name. In either case, it adds a new line to start the description of another change just after the header line of the entry. When `M-x add-change-log-entry` is finished, all is prepared for you to edit in the description of what you changed and how. You must then save the change log file yourself.

The change log file is always visited in Indented Text mode, which means that `LFD` and auto-filling indent each new line like the previous line. This is convenient for entering the contents of an entry, which must all be indented. See Section 20.1 [Text Mode], page 121.

Here is an example of the formatting conventions used in the change log for Emacs:

```
Wed Jun 26 19:29:32 1985  Richard M. Stallman  (rms at mit-prep)

        * xdisp.c (try_window_id):
        If C-k is done at end of next-to-last line,
        this fn updates window_end_vpos and cannot leave
        window_end_pos nonnegative (it is zero, in fact).
```

```
          If display is preempted before lines are output,
          this is inconsistent.  Fix by setting
          blank_end_of_window to nonzero.

Tue Jun 25 05:25:33 1985  Richard M. Stallman  (rms at mit-prep)

          * cmds.c (Fnewline):
          Call the auto fill hook if appropriate.

          * xdisp.c (try_window_id):
          If point is found by compute_motion after xp, record that
          permanently.  If display_text_line sets point position wrong
          (case where line is killed, point is at eob and that line is
          not displayed), set it again in final compute_motion.
```

21.11 Tag Tables

A *tag table* is a description of how a multi-file program is broken up into files. It lists the names of the component files and the names and positions of the functions in each file. Grouping the related files makes it possible to search or replace through all the files with one command. Recording the function names and positions makes possible the `Meta-.` command which you can use to find the definition of a function without having to know which of the files it is in.

Tag tables are stored in files called *tag table files*. The conventional name for a tag table file is '`TAGS`'.

Each entry in the tag table records the name of one tag, the name of the file that the tag is defined in (implicitly), and the position in that file of the tag's definition.

Just what names from the described files are recorded in the tag table depends on the programming language of the described file. They normally include all functions and subroutines, and may also include global variables, data types, and anything else convenient. In any case, each name recorded is called a *tag*.

21.11.1 Source File Tag Syntax

In Lisp code, any function defined with `defun`, any variable defined with `defvar` or `defconst`, and in general the first argument of any expression that starts with '`(def`' in column zero, is a tag.

In C code, any C function is a tag, and so is any typedef if `-t` is specified when the tag table is constructed.

In Fortran code, functions and subroutines are tags.

In LaTeX text, the argument of any of the commands `\chapter`, `\section`, `\subsection`, `\subsubsection`, `\eqno`, `\label`, `\ref`, `\cite`, `\bibitem` and `\typeout` is a tag.

21.11.2 Creating Tag Tables

The `etags` program is used to create a tag table file. It knows the syntax of C, Fortran, LaTeX, Scheme and Emacs Lisp/Common Lisp. To use `etags`, type

`etags` *inputfiles*. . .

as a shell command. It reads the specified files and writes a tag table named '`TAGS`' in the current working directory. `etags` recognizes the language used in an input file based on its file name and contents; there are no switches for specifying the language. The `-t` switch tells `etags` to record typedefs in C code as tags.

If the tag table data become outdated due to changes in the files described in the table, the way to update the tag table is the same way it was made in the first place. It is not necessary to do this often.

If the tag table fails to record a tag, or records it for the wrong file, then Emacs cannot possibly find its definition. However, if the position recorded in the tag table becomes a little bit wrong (due to some editing in the file that the tag definition is in), the only consequence is to slow down finding the tag slightly. Even if the stored position is very wrong, Emacs will still find the tag, but it must search the entire file for it.

So you should update a tag table when you define new tags that you want to have listed, or when you move tag definitions from one file to another, or when changes become substantial. Normally there is no need to update the tag table after each edit, or even every day.

21.11.3 Selecting a Tag Table

Emacs has at any time one *selected* tag table, and all the commands for working with tag tables use the selected one. To select a tag table, type `M-x visit-tags-table`, which reads the tag table file name as an argument. The name '`TAGS`' in the default directory is used as the default file name.

All this command does is store the file name in the variable `tags-file-name`. Emacs does not actually read in the tag table contents until you try to use them. Setting this variable yourself is just as good as using `visit-tags-table`. The variable's initial value is `nil`; this value tells all the commands for working with tag tables that they must ask for a tag table file name to use.

21.11.4 Finding a Tag

The most important thing that a tag table enables you to do is to find the definition of a specific tag.

`M-.` *tag*
: Find first definition of *tag* (`find-tag`).

`C-u M-.`
: Find next alternate definition of last tag specified.

`C-x 4 .` *tag*
: Find first definition of *tag*, but display it in another window (`find-tag-other-window`).

`M-.` (`find-tag`) is the command to find the definition of a specified tag. It searches through the tag table for that tag, as a string, and then uses the tag table info to determine the file that the definition is in and the approximate character position in the file of the definition. Then `find-tag` visits that file, moves point to the approximate character position, and starts searching ever-increasing distances away for the the text that should appear at the beginning of the definition.

If an empty argument is given (just type `RET`), the sexp in the buffer before or around point is used as the name of the tag to find. See Section 21.2 [Lists], page 140, for info on sexps.

The argument to `find-tag` need not be the whole tag name; it can be a substring of a tag name. However, there can be many tag names containing the substring you specify. Since `find-tag` works by searching the text of the tag table, it finds the first tag in the table that the specified substring appears in. The way to find other tags that match the substring is to give `find-tag` a numeric argument, as in `C-u M-.`; this does not read a tag name, but continues searching the tag table's text for another tag containing the same substring last used. If you have a real `META` key, `M-0 M-.` is an easier alternative to `C-u M-.`.

Like most commands that can switch buffers, `find-tag` has another similar command that displays the new buffer in another window. `C-x 4 .` invokes the function `find-tag-other-window`. (This key sequence ends with a period.)

Emacs comes with a tag table file '`TAGS`', in the directory containing Lisp libraries, which includes all the Lisp libraries and all the C sources of Emacs. By specifying this file with `visit-tags-table` and then using `M-.` you can quickly look at the source of any Emacs function.

21.11.5 Searching and Replacing with Tag Tables

The commands in this section visit and search all the files listed in the selected tag table, one by one. For these commands, the tag table serves only to specify a sequence of files to search. A related command is `M-x grep` (see Section 22.1 [Compilation], page 167).

`M-x tags-search`
: Search for the specified regexp through the files in the selected tag table.

`M-x tags-query-replace`
: Perform a `query-replace` on each file in the selected tag table.

`M-,`
: Restart one of the commands above, from the current location of point (`tags-loop-continue`).

`M-x tags-search` reads a regexp using the minibuffer, then visits the files of the selected tag table one by one, and searches through each one for that regexp. It displays the name of the file being searched so you can follow its progress. As soon as an occurrence is found, `tags-search` returns.

Having found one match, you probably want to find all the rest. To find one more match, type `M-,` (`tags-loop-continue`) to resume the `tags-search`. This searches the rest of the current buffer, followed by the remaining files of the tag table.

`M-x tags-query-replace` performs a single `query-replace` through all the files in the tag table. It reads a string to search for and a string to replace with, just like ordinary `M-x query-replace`. It searches much like `M-x tags-search` but repeatedly, processing matches according to your input. See Section 13.7 [Replace], page 78, for more information on `query-replace`.

It is possible to get through all the files in the tag table with a single invocation of `M-x tags-query-replace`. But since any unrecognized character causes the command to exit, you may need to continue where you left off. `M-,` can be used for this. It resumes the last tags search or replace command that you did.

It may have struck you that `tags-search` is a lot like `grep`. You can also run `grep` itself as an inferior of Emacs and have Emacs show you the matching lines one by one. This works mostly the same as running a compilation and having Emacs show you where the errors were. See Section 22.1 [Compilation], page 167.

21.11.6 Stepping Through a Tag Table

If you wish to process all the files in the selected tag table, but `M-x tags-search` and `M-x tags-query-replace` in particular are not what you want, you can use `M-x next-file`.

`C-u M-x next-file`
: With a numeric argument, regardless of its value, visit the first file in the tag table, and prepare to advance sequentially by files.

`M-x next-file`
: Visit the next file in the selected tag table.

21.11.7 Tag Table Inquiries

`M-x list-tags`
: Display a list of the tags defined in a specific program file.

`M-x tags-apropos`
: Display a list of all tags matching a specified regexp.

`M-x list-tags` reads the name of one of the files described by the selected tag table, and displays a list of all the tags defined in that file. The "file name" argument is really just a string to compare against the names recorded in the tag table; it is read as a string rather than as a file name. Therefore, completion and defaulting are not available, and you must enter the string the same way it appears in the tag table. Do not include a directory as part of the file name unless the file name recorded in the tag table includes a directory.

`M-x tags-apropos` is like `apropos` for tags. It reads a regexp, then finds all the tags in the selected tag table whose entries match that regexp, and displays the tag names found.

21.12 Fortran Mode

Fortran mode provides special motion commands for Fortran statements and subprograms, and indentation commands that understand Fortran conventions of nesting, line numbers and continuation statements.

Special commands for comments are provided because Fortran comments are unlike those of other languages.

Built-in abbrevs optionally save typing when you insert Fortran keywords.

Use `M-x fortran-mode` to switch to this major mode. Doing so calls the value of `fortran-mode-hook` as a function of no arguments if that variable has a value that is not `nil`.

Fortran mode was contributed by Michael Prange.

21.12.1 Motion Commands

Fortran mode provides special commands to move by subprograms (functions and subroutines) and by statements. There is also a command to put the region around one subprogram, convenient for killing it or moving it.

`C-M-a` Move to beginning of subprogram (`beginning-of-fortran-subprogram`).

`C-M-e` Move to end of subprogram (`end-of-fortran-subprogram`).

`C-M-h` Put point at beginning of subprogram and mark at end (`mark-fortran-subprogram`).

`C-c C-n` Move to beginning of current or next statement (`fortran-next-statement`).

`C-c C-p` Move to beginning of current or previous statement (`fortran-previous-statement`).

21.12.2 Fortran Indentation

Special commands and features are needed for indenting Fortran code in order to make sure various syntactic entities (line numbers, comment line indicators and continuation line flags) appear in the columns that are required for standard Fortran.

21.12.2.1 Fortran Indentation Commands

`TAB` Indent the current line (`fortran-indent-line`).

`M-LFD` Break the current line and set up a continuation line.

`C-M-q` Indent all the lines of the subprogram point is in (`fortran-indent-subprogram`).

`TAB` is redefined by Fortran mode to reindent the current line for Fortran (`fortran-indent-line`). Line numbers and continuation markers are indented to their required columns, and the body of the statement is independently indented based on its nesting in the program.

The key `C-M-q` is redefined as `fortran-indent-subprogram`, a command to reindent all the lines of the Fortran subprogram (function or subroutine) containing point.

The key `M-LFD` is redefined as `fortran-split-line`, a command to split a line in the appropriate fashion for Fortran. In a non-comment line, the second half becomes a continuation line and is indented accordingly. In a comment line, both halves become separate comment lines.

21.12.2.2 Line Numbers and Continuation

If a number is the first non-whitespace in the line, it is assumed to be a line number and is moved to columns 0 through 4. (Columns are always counted from 0 in GNU Emacs.) If the text on the line starts with the conventional Fortran continuation marker '`$`', it is moved to column 5. If the text begins with any non whitespace character in column 5, it is assumed to be an unconventional continuation marker and remains in column 5.

Line numbers of four digits or less are normally indented one space. This amount is controlled by the variable `fortran-line-number-indent` which is the maximum indentation a line number can have. Line numbers are indented to right-justify them to end in column 4 unless that would require more than this maximum indentation. The default value of the variable is 1.

Simply inserting a line number is enough to indent it according to these rules. As each digit is inserted, the indentation is recomputed. To turn off this feature, set the variable `fortran-electric-line-number` to `nil`. Then inserting line numbers is like inserting anything else.

21.12.2.3 Syntactic Conventions

Fortran mode assumes that you follow certain conventions that simplify the task of understanding a Fortran program well enough to indent it properly:

- Two nested '`do`' loops never share a '`continue`' statement.

- The same character appears in column 5 of all continuation lines, and this character is the value of the variable `fortran-continuation-char`. By default, this character is '`$`'.

If you fail to follow these conventions, the indentation commands may indent some lines unaesthetically. However, a correct Fortran program will retain its meaning when reindented even if the conventions are not followed.

21.12.2.4 Variables for Fortran Indentation

Several additional variables control how Fortran indentation works.

`fortran-do-indent`
: Extra indentation within each level of '`do`' statement (default 3).

`fortran-if-indent`
: Extra indentation within each level of '`if`' statement (default 3).

`fortran-continuation-indent`
: Extra indentation for bodies of continuation lines (default 5).

`fortran-check-all-num-for-matching-do`
: If this is `nil`, indentation assumes that each '`do`' statement ends on a '`continue`' statement. Therefore, when computing indentation for a statement other than '`continue`', it can save time by not checking for a '`do`' statement ending there. If this is non-`nil`, indenting any numbered statement must check for a '`do`' that ends there. The default is `nil`.

`fortran-minimum-statement-indent`
: Minimum indentation for fortran statements. For standard Fortran, this is 6. Statement bodies will never be indented less than this much.

21.12.3 Comments

The usual Emacs comment commands assume that a comment can follow a line of code. In Fortran, the standard comment syntax requires an entire line to be just a comment. Therefore, Fortran mode replaces the standard Emacs comment commands and defines some new variables.

Fortran mode can also handle a nonstandard comment syntax where comments start with '`!`' and can follow other text. Because only some Fortran compilers accept this syntax, Fortran mode will not insert such comments unless you have said in advance to do so. To do this, set the variable `comment-start` to '`"!"`' (see Section 28.2 [Variables], page 218).

`M-;` Align comment or insert new comment (`fortran-comment-indent`).

`C-x ;` Applies to nonstandard '`!`' comments only.

`C-c ;` Turn all lines of the region into comments, or (with arg) turn them back into real code (`fortran-comment-region`).

`M-;` in Fortran mode is redefined as the command `fortran-comment-indent`. Like the usual `M-;` command, this recognizes any kind of existing comment and aligns its text appropriately; if there is no existing comment, a comment is inserted and aligned. But inserting and aligning comments are not the same in Fortran mode as in other modes.

When a new comment must be inserted, if the current line is blank, a full-line comment is inserted. On a non-blank line, a nonstandard '`!`' comment is inserted if you have said you want to use them. Otherwise a full-line comment is inserted on a new line before the current line.

Nonstandard '`!`' comments are aligned like comments in other languages, but full-line comments are different. In a standard full-line comment, the comment delimiter itself must always appear in column zero. What can be aligned is the text within the comment. You can choose from three styles of alignment by setting the variable `fortran-comment-indent-style` to one of these values:

`fixed` The text is aligned at a fixed column, which is the value of `fortran-comment-line-column`. This is the default.

`relative` The text is aligned as if it were a line of code, but with an additional `fortran-comment-line-column` columns of indentation.

`nil` Text in full-line columns is not moved automatically.

In addition, you can specify the character to be used to indent within full-line comments by setting the variable `fortran-comment-indent-char` to the character you want to use.

Fortran mode introduces two variables `comment-line-start` and `comment-line-start-skip` which play for full-line comments the same roles played by `comment-start` and `comment-start-skip` for ordinary text-following comments. Normally these are set properly by Fortran mode so you do not need to change them.

The normal Emacs comment command `C-x ;` has not been redefined. If you use '`!`' comments, this command can be used with them. Otherwise it is useless in Fortran mode.

The command `C-c ;` (`fortran-comment-region`) turns all the lines of the region into comments by inserting the string '`C$$$`' at the front of each one. With a numeric arg, the region is turned back into live code by deleting 'C$$$' from the front of each line in it. The string used for these comments can be controlled by setting the variable `fortran-comment-region`. Note that here we have an example of a command and a variable with the same name; these two uses of the name never conflict because in Lisp and in Emacs it is always clear from the context which one is meant.

21.12.4 Columns

`C-c C-r` Displays a "column ruler" momentarily above the current line (`fortran-column-ruler`).

`C-c C-w` Splits the current window horizontally so that it is 72 columns wide. This may help you avoid going over that limit (`fortran-window-create`).

The command `C-c C-r` (`fortran-column-ruler`) shows a column ruler momentarily above the current line. The comment ruler is two lines of text that show you the locations of columns with special significance in Fortran programs. Square brackets show the limits of the columns for line numbers, and curly brackets show the limits of the columns for the statement body. Column numbers appear above them.

Note that the column numbers count from zero, as always in GNU Emacs. As a result, the numbers may not be those you are familiar with; but the actual positions in the line are standard Fortran.

The text used to display the column ruler is the value of the variable `fortran-comment-ruler`. By changing this variable, you can change the display.

For even more help, use `C-c C-w` (`fortran-window-create`), a command which splits the current window horizontally, making a window 72 columns wide. By editing in this window you can immediately see when you make a line too wide to be correct Fortran.

21.12.5 Fortran Keyword Abbrevs

Fortran mode provides many built-in abbrevs for common keywords and declarations. These are the same sort of abbrev that you can define yourself. To use them, you must turn on Abbrev mode. see Chapter 23 [Abbrevs], page 177.

The built-in abbrevs are unusual in one way: they all start with a semicolon. You cannot normally use semicolon in an abbrev, but Fortran mode makes this possible by changing the syntax of semicolon to "word constituent".

For example, one built-in Fortran abbrev is '`;c`' for '`continue`'. If you insert '`;c`' and then insert a punctuation character such as a space or a newline, the '`;c`' will change automatically to '`continue`', provided Abbrev mode is enabled.

Type '`;?`' or '`;C-h`' to display a list of all the built-in Fortran abbrevs and what they stand for.

22 Compiling and Testing Programs

The previous chapter discusses the Emacs commands that are useful for making changes in programs. This chapter deals with commands that assist in the larger process of developing and maintaining programs.

22.1 Running 'make', or Compilers Generally

Emacs can run compilers for noninteractive languages such as C and Fortran as inferior processes, feeding the error log into an Emacs buffer. It can also parse the error messages and visit the files in which errors are found, moving point right to the line where the error occurred.

`M-x compile`
: Run a compiler asynchronously under Emacs, with error messages to '`*compilation*`' buffer.

`M-x grep`
: Run `grep` asynchronously under Emacs, with matching lines listed in the buffer named '`*compilation*`'.

`M-x kill-compiler`
`M-x kill-grep`
: Kill the running compilation or `grep` subprocess.

`` C-x ` ``
: Visit the locus of the next compiler error message or `grep` match.

To run `make` or another compiler, do `M-x compile`. This command reads a shell command line using the minibuffer, and then executes the specified command line in an inferior shell with output going to the buffer named '`*compilation*`'. The current buffer's default directory is used as the working directory for the execution of the command; normally, therefore, the makefile comes from this directory.

When the shell command line is read, the minibuffer appears containing a default command line, which is the command you used the last time you did `M-x compile`. If you type just `RET`, the same command line is used again. The first `M-x compile` provides `make -k` as the default. The default is taken from the variable `compile-command`; if the appropriate compilation command for a file is something other than `make -k`, it can be useful to have the file specify a local value for `compile-command` (see Section 28.2.4 [File Variables], page 221).

Starting a compilation causes the buffer '`*compilation*`' to be displayed in another window but not selected. Its mode line tells you whether compilation is finished, with the word '`run`' or 'exit' inside the parentheses. You do not have to keep this buffer visible; compilation continues in any case.

To kill the compilation process, do M-x kill-compilation. You will see that the mode line of the '*compilation*' buffer changes to say 'signal' instead of 'run'. Starting a new compilation also kills any running compilation, as only one can exist at any time. However, this requires confirmation before actually killing a compilation that is running.

To parse the compiler error messages, type C-x ` (next-error). The character following the C-x is the grave accent, not the single quote. This command displays the buffer '*compilation*' in one window and the buffer in which the next error occurred in another window. Point in that buffer is moved to the line where the error was found. The corresponding error message is scrolled to the top of the window in which '*compilation*' is displayed.

The first time C-x ` is used after the start of a compilation, it parses all the error messages, visits all the files that have error messages, and makes markers pointing at the lines that the error messages refer to. Then it moves to the first error message location. Subsequent uses of C-x ` advance down the data set up by the first use. When the preparsed error messages are exhausted, the next C-x ` checks for any more error messages that have come in; this is useful if you start editing the compiler errors while the compilation is still going on. If no more error messages have come in, C-x ` reports an error.

C-u C-x ` discards the preparsed error message data and parses the '*compilation*' buffer over again, then displaying the first error. This way, you can process the same set of errors again.

Instead of running a compiler, you can run grep and see the lines on which matches were found. To do this, type M-x grep with an argument line that contains the same arguments you would give grep when running it normally: a grep-style regexp (usually in singlequotes to quote the shell's special characters) followed by filenames which may use wildcards. The output from grep goes in the '*compilation*' buffer and the lines that matched can be found with C-x ` as if they were compilation errors.

Note: a shell is used to run the compile command, but the shell is told that it should be noninteractive. This means in particular that the shell starts up with no prompt. If you find your usual shell prompt making an unsightly appearance in the '*compilation*' buffer, it means you have made a mistake in your shell's init file ('.cshrc' or '.shrc' or ...) by setting the prompt unconditionally. The shell init file should set the prompt only if there already is a prompt. In csh, here is how to do it:

```
if ($?prompt) set prompt = ...
```

22.2 Major Modes for Lisp

Emacs has four different major modes for Lisp. They are the same in terms of editing commands, but differ in the commands for executing Lisp expressions.

Emacs-Lisp mode
: The mode for editing source files of programs to run in Emacs Lisp. This mode defines `C-M-x` to evaluate the current defun. See Section 22.3 [Lisp Libraries], page 169.

Lisp Interaction mode
: The mode for an interactive session with Emacs Lisp. It defines `LFD` to evaluate the sexp before point and insert its value in the buffer. See Section 22.6 [Lisp Interaction], page 175.

Lisp mode
: The mode for editing source files of programs that run in Lisps other than Emacs Lisp. This mode defines `C-M-x` to send the current defun to an inferior Lisp process. See Section 22.7 [External Lisp], page 175.

Inferior Lisp mode
: The mode for an interactive session with an inferior Lisp process. This mode combines the special features of Lisp mode and Shell mode (see Section 27.4.3 [Shell Mode], page 213).

Scheme mode
: Like Lisp mode but for Scheme programs.

Inferior Scheme mode
: The mode for an interactive session with an inferior Scheme process.

22.3 Libraries of Lisp Code for Emacs

Lisp code for Emacs editing commands is stored in files whose names conventionally end in '`.el`'. This ending tells Emacs to edit them in Emacs-Lisp mode (see Section 22.2 [Lisp Modes], page 169).

22.3.1 Loading Libraries

To execute a file of Emacs Lisp, use `M-x load-file`. This command reads a file name using the minibuffer and then executes the contents of that file as Lisp code. It is not necessary to visit the file first; in any case, this command reads the file as found on disk, not text in an Emacs buffer.

Once a file of Lisp code is installed in the Emacs Lisp library directories, users can load it using `M-x load-library`. Programs can load it by calling `load-library`, or with `load`, a more primitive function that is similar but accepts some additional arguments.

`M-x load-library` differs from `M-x load-file` in that it searches a sequence of directories and tries three file names in each directory. The three names are, first, the specified name with '`.elc`' appended; second, with '`.el`' appended; third, the specified name alone. A '`.elc`' file would be the result of compiling the Lisp file into byte code; it is loaded if possible in preference to the Lisp file itself because the compiled file will load and run faster.

Because the argument to `load-library` is usually not in itself a valid file name, file name completion is not available. Indeed, when using this command, you usually do not know exactly what file name will be used.

The sequence of directories searched by `M-x load-library` is specified by the variable `load-path`, a list of strings that are directory names. The default value of the list contains the directory where the Lisp code for Emacs itself is stored. If you have libraries of your own, put them in a single directory and add that directory to `load-path`. `nil` in this list stands for the current default directory, but it is probably not a good idea to put `nil` in the list. If you find yourself wishing that `nil` were in the list, most likely what you really want to do is use `M-x load-file` this once.

Often you do not have to give any command to load a library, because the commands defined in the library are set up to *autoload* that library. Running any of those commands causes `load` to be called to load the library; this replaces the autoload definitions with the real ones from the library.

If autoloading a file does not finish, either because of an error or because of a `C-g` quit, all function definitions made by the file are undone automatically. So are any calls to `provide`. As a consequence, if you use one of the autoloadable commands again, the entire file will be loaded a second time. This prevents problems where the command is no longer autoloading but it works wrong because not all the file was loaded. Function definitions are undone only for autoloading; explicit calls to `load` do not undo anything if loading is not completed.

22.3.2 Compiling Libraries

Emacs Lisp code can be compiled into byte-code which loads faster, takes up less space when loaded, and executes faster.

The way to make a byte-code compiled file from an Emacs-Lisp source file is with `M-x byte-compile-file`. The default argument for this function is the file visited in the current buffer. It reads the specified file, compiles it into byte code, and writes an output file whose name is made by appending ‘`c`’ to the input file name. Thus, the file ‘`rmail.el`’ would be compiled into ‘`rmail.elc`’.

To recompile the changed Lisp files in a directory, use `M-x byte-recompile-directory`. Specify just the directory name as an argument. Each ‘`.el`’ file that has been byte-compiled before is byte-compiled again if it has changed since the previous compilation. A numeric argument to this command tells it to offer to compile each ‘`.el`’ file that has not already been compiled. You must answer `y` or `n` to each offer.

Emacs can be invoked noninteractively from the shell to do byte compilation with the aid of the function `batch-byte-compile`. In this case, the files to be compiled are specified with command-line arguments. Use a shell command of the form

```
emacs -batch -f batch-byte-compile files...
```

Directory names may also be given as arguments; `byte-recompile-directory` is invoked (in effect) on each such directory. `batch-byte-compile` uses all the remaining command-line arguments as file or directory names, then kills the Emacs process.

`M-x disassemble` explains the result of byte compilation. Its argument is a function name. It displays the byte-compiled code in a help window in symbolic form, one instruction per line. If the instruction refers to a variable or constant, that is shown too.

22.3.3 Converting Mocklisp to Lisp

GNU Emacs can run Mocklisp files by converting them to Emacs Lisp first. To convert a Mocklisp file, visit it and then type `M-x convert-mocklisp-buffer`. Then save the resulting buffer of Lisp file in a file whose name ends in ‘`.el`’ and use the new file as a Lisp library.

It does not currently work to byte-compile converted Mocklisp code. This is because converted Mocklisp code uses some special Lisp features to deal with Mocklisp’s incompatible ideas of how arguments are evaluated and which values signify “true” or “false”.

22.4 Evaluating Emacs-Lisp Expressions

Lisp programs intended to be run in Emacs should be edited in Emacs-Lisp mode; this will happen automatically for file names ending in '`.el`'. By contrast, Lisp mode itself is used for editing Lisp programs intended for other Lisp systems. Emacs-Lisp mode can be selected with the command `M-x emacs-lisp-mode`.

For testing of Lisp programs to run in Emacs, it is useful to be able to evaluate part of the program as it is found in the Emacs buffer. For example, after changing the text of a Lisp function definition, evaluating the definition installs the change for future calls to the function. Evaluation of Lisp expressions is also useful in any kind of editing task for invoking noninteractive functions (functions that are not commands).

`M-ESC`
: Read a Lisp expression in the minibuffer, evaluate it, and print the value in the minibuffer (`eval-expression`).

`C-x C-e`
: Evaluate the Lisp expression before point, and print the value in the minibuffer (`eval-last-sexp`).

`C-M-x`
: Evaluate the defun containing or after point, and print the value in the minibuffer (`eval-defun`).

`M-x eval-region`
: Evaluate all the Lisp expressions in the region.

`M-x eval-current-buffer`
: Evaluate all the Lisp expressions in the buffer.

`M-ESC` (`eval-expression`) is the most basic command for evaluating a Lisp expression interactively. It reads the expression using the minibuffer, so you can execute any expression on a buffer regardless of what the buffer contains. When the expression is evaluated, the current buffer is once again the buffer that was current when `M-ESC` was typed.

`M-ESC` can easily confuse users who do not understand it, especially on keyboards with autorepeat where it can result from holding down the `ESC` key for too long. Therefore, `eval-expression` is normally a disabled command. Attempting to use this command asks for confirmation and gives you the option of enabling it; once you enable the command, confirmation will no longer be required for it. See Section 28.4.3 [Disabling], page 229.

In Emacs-Lisp mode, the key `C-M-x` is bound to the function `eval-defun`, which parses the defun containing or following point as a Lisp expression and evaluates it. The value is printed in the echo area. This command is convenient for installing in the Lisp environment changes that you have just made in the text of a function definition.

The command `C-x C-e` (`eval-last-sexp`) performs a similar job but is available in all major modes, not just Emacs-Lisp mode. It finds the sexp before point, reads it as a Lisp expression, evaluates it, and prints the value in the echo area. It is sometimes useful to type in an expression and then, with point still after it, type `C-x C-e`.

If `C-M-x` or `C-x C-e` is given a numeric argument, it prints the value by insertion into the current buffer at point, rather than in the echo area. The argument value does not matter.

The most general command for evaluating Lisp expressions from a buffer is `eval-region`. `M-x eval-region` parses the text of the region as one or more Lisp expressions, evaluating them one by one. `M-x eval-current-buffer` is similar but evaluates the entire buffer. This is a reasonable way to install the contents of a file of Lisp code that you are just ready to test. After finding and fixing a bug, use `C-M-x` on each function that you change, to keep the Lisp world in step with the source file.

22.5 The Emacs-Lisp Debugger

GNU Emacs contains a debugger for Lisp programs executing inside it. This debugger is normally not used; many commands frequently get Lisp errors when invoked in inappropriate contexts (such as `C-f` at the end of the buffer) and it would be very unpleasant for that to enter a special debugging mode. When you want to make Lisp errors invoke the debugger, you must set the variable `debug-on-error` to non-`nil`. Quitting with `C-g` is not considered an error, and `debug-on-error` has no effect on the handling of `C-g`. However, if you set `debug-on-quit` non-`nil`, `C-g` will invoke the debugger. This can be useful for debugging an infinite loop; type `C-g` once the loop has had time to reach its steady state. `debug-on-quit` has no effect on errors.

You can also cause the debugger to be entered when a specified function is called, or at a particular place in Lisp code. Use `M-x debug-on-entry` with argument *fun-name* to cause function *fun-name* to enter the debugger as soon as it is called. Use `M-x cancel-debug-on-entry` to make the function stop entering the debugger when called. (Redefining the function also does this.) To enter the debugger from some other place in Lisp code, you must insert the expression `(debug)` there and install the changed code with `C-M-x`. See Section 22.4 [Lisp Eval], page 172.

When the debugger is entered, it displays the previously selected buffer in one window and a buffer named '`*Backtrace*`' in another window. The backtrace buffer contains one line for each level of Lisp function execution currently going on. At the beginning of this buffer is a message describing the reason that the debugger was invoked (such as, what error message if it was invoked due to an error).

The backtrace buffer is read-only, and is in a special major mode, Backtrace mode, in which letters are defined as debugger commands. The usual Emacs editing commands are available; you can switch windows to examine the buffer that was being edited at the time of the error, and you can also switch buffers, visit files, and do any other sort of editing. However, the debugger is a recursive editing level (see Section 27.1 [Recursive Edit], page 207) and it is wise to go back to the backtrace buffer and exit the debugger officially when you don't want to use it any more. Exiting the debugger kills the backtrace buffer.

The contents of the backtrace buffer show you the functions that are executing and the arguments that were given to them. It has the additional purpose of allowing you to specify a stack frame by moving point to the line describing that frame. The frame whose line point is on is considered the *current frame*. Some of the debugger commands operate on the current frame. Debugger commands are mainly used for stepping through code an expression at a time. Here is a list of them.

`c`
: Exit the debugger and continue execution. In most cases, execution of the program continues as if the debugger had never been entered (aside from the effect of any variables or data structures you may have changed while inside the debugger). This includes entry to the debugger due to function entry or exit, explicit invocation, quitting or certain errors. Most errors cannot be continued; trying to continue one of them causes the same error to occur again.

`d`
: Continue execution, but enter the debugger the next time a Lisp function is called. This allows you to step through the subexpressions of an expression, seeing what values the subexpressions compute and what else they do.

  The stack frame made for the function call which enters the debugger in this way will be flagged automatically for the debugger to be called when the frame is exited. You can use the `u` command to cancel this flag.

`b`
: Set up to enter the debugger when the current frame is exited. Frames that will invoke the debugger on exit are flagged with stars.

`u`
: Don't enter the debugger when the current frame is exited. This cancels a `b` command on that frame.

`e`
: Read a Lisp expression in the minibuffer, evaluate it, and print the value in the echo area. This is the same as the command `M-ESC`, except that `e` is not normally disabled like `M-ESC`.

`q`
: Terminate the program being debugged; return to top-level Emacs command execution.

  If the debugger was entered due to a `C-g` but you really want to quit, not to debug, use the `q` command.

`r`
: Return a value from the debugger. The value is computed by reading an expression with the minibuffer and evaluating it.

  The value returned by the debugger makes a difference when the debugger was invoked due to exit from a Lisp call frame (as requested with `b`); then the value specified in the `r` command is used as the value of that frame.

The debugger's return value also matters with many errors. For example, `wrong-type-argument` errors will use the debugger's return value instead of the invalid argument; `no-catch` errors will use the debugger value as a throw tag instead of the tag that was not found. If an error was signaled by calling the Lisp function `signal`, the debugger's return value is returned as the value of `signal`.

22.6 Lisp Interaction Buffers

The buffer '`*scratch*`' which is selected when Emacs starts up is provided for evaluating Lisp expressions interactively inside Emacs. Both the expressions you evaluate and their output goes in the buffer.

The '`*scratch*`' buffer's major mode is Lisp Interaction mode, which is the same as Emacs-Lisp mode except for one command, `LFD`. In Emacs-Lisp mode, `LFD` is an indentation command, as usual. In Lisp Interaction mode, `LFD` is bound to `eval-print-last-sexp`. This function reads the Lisp expression before point, evaluates it, and inserts the value in printed representation before point.

Thus, the way to use the '`*scratch*`' buffer is to insert Lisp expressions at the end, ending each one with `LFD` so that it will be evaluated. The result is a complete typescript of the expressions you have evaluated and their values.

The rationale for this feature is that Emacs must have a buffer when it starts up, but that buffer is not useful for editing files since a new buffer is made for every file that you visit. The Lisp interpreter typescript is the most useful thing I can think of for the initial buffer to do. `M-x lisp-interaction-mode` will put any buffer in Lisp Interaction mode.

22.7 Running an External Lisp

Emacs has facilities for running programs in other Lisp systems. You can run a Lisp process as an inferior of Emacs, and pass expressions to it to be evaluated. You can also pass changed function definitions directly from the Emacs buffers in which you edit the Lisp programs to the inferior Lisp process.

To run an inferior Lisp process, type `M-x run-lisp`. This runs the program named `lisp`, the same program you would run by typing `lisp` as a shell command, with both input and output going through an Emacs buffer named '`*lisp*`'. That is to say, any "terminal output" from Lisp will go into the buffer, advancing point, and any "terminal input" for Lisp comes from text in the buffer. To give input to Lisp, go to the end of the buffer and type the input, terminated by `RET`. The '`*lisp*`' buffer is in Inferior Lisp mode, a mode which has all the special characteristics of Lisp mode and Shell mode (see Section 27.4.3 [Shell Mode], page 213).

For the source files of programs to run in external Lisps, use Lisp mode. This mode can be selected with `M-x lisp-mode`, and is used automatically for files whose names end in '`.l`' or '`.lisp`', as most Lisp systems usually expect.

When you edit a function in a Lisp program you are running, the easiest way to send the changed definition to the inferior Lisp process is the key `C-M-x`. In Lisp mode, this runs the function `lisp-send-defun`, which finds the defun around or following point and sends it as input to the Lisp process. (Emacs can send input to any inferior process regardless of what buffer is current.)

Contrast the meanings of `C-M-x` in Lisp mode (for editing programs to be run in another Lisp system) and Emacs-Lisp mode (for editing Lisp programs to be run in Emacs): in both modes it has the effect of installing the function definition that point is in, but the way of doing so is different according to where the relevant Lisp environment is found. See Section 22.2 [Lisp Modes], page 169.

23 Abbrevs

An *abbrev* is a word which *expands*, if you insert it, into some different text. Abbrevs are defined by the user to expand in specific ways. For example, you might define '`foo`' as an abbrev expanding to '`find outer otter`'. With this abbrev defined, you would be able to get '`find outer otter `' into the buffer by typing `f o o SPC`.

Abbrevs expand only when Abbrev mode (a minor mode) is enabled. Disabling Abbrev mode does not cause abbrev definitions to be forgotten, but they do not expand until Abbrev mode is enabled again. The command `M-x abbrev-mode` toggles Abbrev mode; with a numeric argument, it turns Abbrev mode on if the argument is positive, off otherwise. See Section 28.1 [Minor Modes], page 217. `abbrev-mode` is also a variable; Abbrev mode is on when the variable is non-`nil`. The variable `abbrev-mode` automatically becomes local to the current buffer when it is set.

Abbrev definitions can be *mode-specific*—active only in one major mode. Abbrevs can also have *global* definitions that are active in all major modes. The same abbrev can have a global definition and various mode-specific definitions for different major modes. A mode specific definition for the current major mode overrides a global definition.

Abbrevs can be defined interactively during the editing session. Lists of abbrev definitions can also be saved in files and reloaded in later sessions. Some users keep extensive lists of abbrevs that they load in every session.

A second kind of abbreviation facility is called the *dynamic expansion*. Dynamic abbrev expansion happens only when you give an explicit command and the result of the expansion depends only on the current contents of the buffer. See Section 23.5 [Dynamic Abbrevs], page 181.

23.1 Defining Abbrevs

`C-x +` Define an abbrev to expand into some text before point (`add-global-abbrev`).

`C-x C-a` Similar, but define an abbrev available only in the current major mode (`add-mode-abbrev`).

`C-x -` Define a word in the buffer as an abbrev (`inverse-add-global-abbrev`).

`C-x C-h` Define a word in the buffer as a mode-specific abbrev (`inverse-add-mode-abbrev`).

`M-x kill-all-abbrevs`
: After this command, there are no abbrev definitions in effect.

The usual way to define an abbrev is to enter the text you want the abbrev to expand to, position point after it, and type `C-x +` (`add-global-abbrev`). This reads the abbrev itself using the minibuffer, and then defines it as an abbrev for one or more words before point. Use a numeric argument to say how many words before point should be taken as the expansion. For example, to define the abbrev '`foo`' as mentioned above, insert the text '`find outer otter`' and then type `C-u 3 C-x + f o o RET`.

An argument of zero to `C-x +` means to use the contents of the region as the expansion of the abbrev being defined.

The command `C-x C-a` (`add-mode-abbrev`) is similar, but defines a mode-specific abbrev. Mode specific abbrevs are active only in a particular major mode. `C-x C-a` defines an abbrev for the major mode in effect at the time `C-x C-a` is typed. The arguments work the same as for `C-x +`.

If the text of the abbrev you want is already in the buffer instead of the expansion, use command `C-x -` (`inverse-add-global-abbrev`) instead of `C-x +`, or use `C-x C-h` (`inverse-add-mode-abbrev`) instead of `C-x C-a`. These commands are called "inverse" because they invert the meaning of the argument found in the buffer and the argument read using the minibuffer.

To change the definition of an abbrev, just add the new definition. You will be asked to confirm if the abbrev has a prior definition. To remove an abbrev definition, give a negative argument to `C-x +` or `C-x C-a`. You must choose the command to specify whether to kill a global definition or a mode-specific definition for the current mode, since those two definitions are independent for one abbrev.

`M-x kill-all-abbrevs` removes all the abbrev definitions there are.

23.2 Controlling Abbrev Expansion

An abbrev expands whenever it is present in the buffer just before point and a self-inserting punctuation character (`SPC`, comma, etc.) is typed. Most often the way an abbrev is used is to insert the abbrev followed by punctuation.

Abbrev expansion preserves case; thus, '`foo`' expands into '`find outer otter`'; '`Foo`' into '`Find outer otter`', and '`FOO`' into '`FIND OUTER OTTER`' or '`Find Outer Otter`' according to the variable `abbrev-all-caps` (a non-`nil` value chooses the first of the two expansions).

These two commands are used to control abbrev expansion:

`M-'` Separate a prefix from a following abbrev to be expanded (`abbrev-prefix-mark`).

`C-x '` Expand the abbrev before point (`expand-abbrev`). This is effective even when Abbrev mode is not enabled.

`M-x unexpand-abbrev`
Undo last abbrev expansion.

`M-x expand-region-abbrevs`
Expand some or all abbrevs found in the region.

You may wish to expand an abbrev with a prefix attached; for example, if '`cnst`' expands into '`construction`', you might want to use it to enter '`reconstruction`'. It does not work to type `recnst`, because that is not necessarily a defined abbrev. What does work is to use the command `M-'` (`abbrev-prefix-mark`) in between the prefix '`re`' and the abbrev '`cnst`'. First, insert '`re`'. Then type `M-'`; this inserts a minus sign in the buffer to indicate that it has done its work. Then insert the abbrev '`cnst`'; the buffer now contains '`re-cnst`'. Now insert a punctuation character to expand the abbrev '`cnst`' into '`construction`'. The minus sign is deleted at this point, because `M-'` left word for this to be done. The resulting text is the desired '`reconstruction`'.

If you actually want the text of the abbrev in the buffer, rather than its expansion, you can accomplish this by inserting the following punctuation with `C-q`. Thus, `foo C-q -` leaves '`foo-`' in the buffer.

If you expand an abbrev by mistake, you can undo the expansion (replace the expansion by the original abbrev text) with `M-x unexpand-abbrev`. `C-_` (`undo`) can also be used to undo the expansion; but first it will undo the insertion of the following punctuation character!

`M-x expand-region-abbrevs` searches through the region for defined abbrevs, and for each one found offers to replace it with its expansion. This command is useful if you have typed in text using abbrevs but forgot to turn on Abbrev mode first. It may also be useful together with a special set of abbrev definitions for making several global replacements at once. This command is effective even if Abbrev mode is not enabled.

23.3 Examining and Editing Abbrevs

`M-x list-abbrevs`
Print a list of all abbrev definitions.

`M-x edit-abbrevs`
Edit a list of abbrevs; you can add, alter or remove definitions.

The output from M-x list-abbrevs looks like this:

```
(lisp-mode-abbrev-table)
"dk"        0    "define-key"
(global-abbrev-table)
"dfn"        0    "definition"
```

(Some blank lines of no semantic significance, and some other abbrev tables, have been omitted.)

A line containing a name in parentheses is the header for abbrevs in a particular abbrev table; global-abbrev-table contains all the global abbrevs, and the other abbrev tables that are named after major modes contain the mode-specific abbrevs.

Within each abbrev table, each nonblank line defines one abbrev. The word at the beginning is the abbrev. The number that appears is the number of times the abbrev has been expanded. Emacs keeps track of this to help you see which abbrevs you actually use, in case you decide to eliminate those that you don't use often. The string at the end of the line is the expansion.

M-x edit-abbrevs allows you to add, change or kill abbrev definitions by editing a list of them in an Emacs buffer. The list has the same format described above. The buffer of abbrevs is called '*Abbrevs*', and is in Edit-Abbrevs mode. This mode redefines the key C-c C-c to install the abbrev definitions as specified in the buffer. The command that does this is edit-abbrevs-redefine. Any abbrevs not described in the buffer are eliminated when this is done.

edit-abbrevs is actually the same as list-abbrevs except that it selects the buffer '*Abbrevs*' whereas list-abbrevs merely displays it in another window.

23.4 Saving Abbrevs

These commands allow you to keep abbrev definitions between editing sessions.

M-x write-abbrev-file
: Write a file describing all defined abbrevs.

M-x read-abbrev-file
: Read such a file and define abbrevs as specified there.

M-x quietly-read-abbrev-file
: Similar but do not display a message about what is going on.

`M-x define-abbrevs`
: Define abbrevs from buffer.

`M-x insert-abbrevs`
: Insert all abbrevs and their expansions into the buffer.

`M-x write-abbrev-file` reads a file name using the minibuffer and writes a description of all current abbrev definitions into that file. The text stored in the file looks like the output of `M-x list-abbrevs`. This is used to save abbrev definitions for use in a later session.

`M-x read-abbrev-file` reads a file name using the minibuffer and reads the file, defining abbrevs according to the contents of the file. `M-x quietly-read-abbrev-file` is the same except that it does not display a message in the echo area saying that it is doing its work; it is actually useful primarily in the '`.emacs`' file. If an empty argument is given to either of these functions, the file name used is the value of the variable `abbrev-file-name`, which is by default `"~/.abbrev_defs"`.

Emacs will offer to save abbrevs automatically if you have changed any of them, whenever it offers to save all files (for `C-x s` or `C-x C-c`). This feature can be inhibited by setting the variable `save-abbrevs` to `nil`.

The commands `M-x insert-abbrevs` and `M-x define-abbrevs` are similar to the previous commands but work on text in an Emacs buffer. `M-x insert-abbrevs` inserts text into the current buffer before point, describing all current abbrev definitions; `M-x define-abbrevs` parses the entire current buffer and defines abbrevs accordingly.

23.5 Dynamic Abbrev Expansion

The abbrev facility described above operates automatically as you insert text, but all abbrevs must be defined explicitly. By contrast, *dynamic abbrevs* allow the meanings of abbrevs to be determined automatically from the contents of the buffer, but dynamic abbrev expansion happens only when you request it explicitly.

`M-/`
: Expand the word in the buffer before point as a *dynamic abbrev*, by searching in the buffer for words starting with that abbreviation (`dabbrev-expand`).

For example, if the buffer contains '`does this follow `' and you type `f o M-/`, the effect is to insert '`follow`' because that is the last word in the buffer that starts with '`fo`'. A numeric argument to `M-/` says to take the second, third, etc. distinct expansion found looking backward from point.

Repeating `M-/` searches for an alternative expansion by looking farther back. After the entire buffer before point has been considered, the buffer after point is searched.

Dynamic abbrev expansion is completely independent of Abbrev mode; the expansion of a word with `M-/` is completely independent of whether it has a definition as an ordinary abbrev.

24 Editing Pictures

If you want to create a picture made out of text characters (for example, a picture of the division of a register into fields, as a comment in a program), use the command `edit-picture` to enter Picture mode.

In Picture mode, editing is based on the *quarter-plane* model of text, according to which the text characters lie studded on an area that stretches infinitely far to the right and downward. The concept of the end of a line does not exist in this model; the most you can say is where the last nonblank character on the line is found.

Of course, Emacs really always considers text as a sequence of characters, and lines really do have ends. But in Picture mode most frequently-used keys are rebound to commands that simulate the quarter-plane model of text. They do this by inserting spaces or by converting tabs to spaces.

Most of the basic editing commands of Emacs are redefined by Picture mode to do essentially the same thing but in a quarter-plane way. In addition, Picture mode defines various keys starting with the `C-c` prefix to run special picture editing commands.

One of these keys, `C-c C-c`, is pretty important. Often a picture is part of a larger file that is usually edited in some other major mode. `M-x edit-picture` records the name of the previous major mode, and then you can use the `C-c C-c` command (`picture-mode-exit`) to restore that mode. `C-c C-c` also deletes spaces from the ends of lines, unless given a numeric argument.

The commands used in Picture mode all work in other modes (provided the '`picture`' library is loaded), but are not bound to keys except in Picture mode. Note that the descriptions below talk of moving "one column" and so on, but all the picture mode commands handle numeric arguments as their normal equivalents do.

Turning on Picture mode calls the value of the variable `picture-mode-hook` as a function, with no arguments, if that value exists and is non-`nil`.

24.1 Basic Editing in Picture Mode

Most keys do the same thing in Picture mode that they usually do, but do it in a quarter-plane style. For example, `C-f` is rebound to run `picture-forward-column`, which is defined to move point one column to the right, by inserting a space if necessary, so that the actual end of the line makes no difference. C-b is rebound to run picture-backward-column, which always moves point left one column, converting a tab to multiple spaces if necessary. C-n and C-p are rebound to run picture-move-down and picture-move-up, which can either insert spaces or convert tabs as necessary to make sure that point stays in exactly the same column. C-e runs picture-end-of-line, which moves to after the last nonblank character on the line. There is no need to change C-a, as the choice of screen model does not affect beginnings of lines.

Insertion of text is adapted to the quarter-plane screen model through the use of Overwrite mode (see Section 28.1 [Minor Modes], page 217). Self-inserting characters replace existing text, column by column, rather than pushing existing text to the right. RET runs picture-newline, which just moves to the beginning of the following line so that new text will replace that line.

Deletion and killing of text are replaced with erasure. DEL (picture-backward-clear-column) replaces the preceding character with a space rather than removing it. C-d (picture-clear-column) does the same thing in a forward direction. C-k (picture-clear-line) really kills the contents of lines, but does not ever remove the newlines from the buffer.

To do actual insertion, you must use special commands. C-o (picture-open-line) still creates a blank line, but does so after the current line; it never splits a line. C-M-o, split-line, makes sense in Picture mode, so it is not changed. LFD (picture-duplicate-line) inserts below the current line another line with the same contents.

Real deletion can be done with C-w, or with C-c C-d (which is defined as delete-char, as C-d is in other modes), or with one of the picture rectangle commands (see Section 24.4 [Rectangles in Picture], page 185).

24.2 Controlling Motion after Insert

Since "self-inserting" characters in Picture mode just overwrite and move point, there is no essential restriction on how point should be moved. Normally point moves right, but you can specify any of the eight orthogonal or diagonal directions for motion after a "self-inserting" character. This is useful for drawing lines in the buffer.

C-c < Move left after insertion (picture-movement-left).

C-c > Move right after insertion (picture-movement-right).

C-c ^ Move up after insertion (picture-movement-up).

C-c . Move down after insertion (picture-movement-down).

`C-c `` Move up and left ("northwest") after insertion (`picture-movement-nw`).

`C-c '` Move up and right ("northeast") after insertion (`picture-movement-ne`).

`C-c /` Move down and left ("southwest") after insertion (`picture-movement-sw`).

`C-c \` Move down and right ("southeast") after insertion (`picture-movement-se`).

Two motion commands move based on the current Picture insertion direction. The command `C-c C-f` (`picture-motion`) moves in the same direction as motion after "insertion" currently does, while `C-c C-b` (`picture-motion-reverse`) moves in the opposite direction.

24.3 Picture Mode Tabs

Two kinds of tab-like action are provided in Picture mode. Context-based tabbing is done with `M-TAB` (`picture-tab-search`). With no argument, it moves to a point underneath the next "interesting" character that follows whitespace in the previous nonblank line. "Next" here means "appearing at a horizontal position greater than the one point starts out at". With an argument, as in `C-u M-TAB`, this command moves to the next such interesting character in the current line. `M-TAB` does not change the text; it only moves point. "Interesting" characters are defined by the variable `picture-tab-chars`, which contains a string whose characters are all considered interesting. Its default value is `"!-~"`.

`TAB` itself runs `picture-tab`, which operates based on the current tab stop settings; it is the Picture mode equivalent of `tab-to-tab-stop`. Normally it just moves point, but with a numeric argument it clears the text that it moves over.

The context-based and tab-stop-based forms of tabbing are brought together by the command `C-c TAB`, `picture-set-tab-stops`. This command sets the tab stops to the positions which `M-TAB` would consider significant in the current line. The use of this command, together with `TAB`, can get the effect of context-based tabbing. But `M-TAB` is more convenient in the cases where it is sufficient.

24.4 Picture Mode Rectangle Commands

Picture mode defines commands for working on rectangular pieces of the text in ways that fit with the quarter-plane model. The standard rectangle commands may also be useful (see Section 10.4 [Rectangles], page 59).

`C-c C-k` Clear out the region-rectangle (`picture-clear-rectangle`). With argument, kill it.

`C-c C-w` *r* Similar but save rectangle contents in register *r* first (`picture-clear-rectangle-to-register`).

`C-c C-y` Copy last killed rectangle into the buffer by overwriting, with upper left corner at point (`picture-yank-rectangle`). With argument, insert instead.

`C-c C-x` *r* Similar, but use the rectangle in register *r* (`picture-yank-rectangle-from-register`).

The picture rectangle commands `C-c C-k` (`picture-clear-rectangle`) and `C-c C-w` (`picture-clear-rectangle-to-register`) differ from the standard rectangle commands in that they normally clear the rectangle instead of deleting it; this is analogous with the way `C-d` is changed in Picture mode.

However, deletion of rectangles can be useful in Picture mode, so these commands delete the rectangle if given a numeric argument.

The Picture mode commands for yanking rectangles differ from the standard ones in overwriting instead of inserting. This is the same way that Picture mode insertion of other text is different from other modes. `C-c C-y` (`picture-yank-rectangle`) inserts (by overwriting) the rectangle that was most recently killed, while `C-c C-x` (`picture-yank-rectangle-from-register`) does likewise for the rectangle found in a specified register.

25 Sending Mail

To send a message in Emacs, you start by typing a command (`C-x m`) to select and initialize the '`*mail*`' buffer. Then you edit the text and headers of the message in this buffer, and type another command (`C-c C-c`) to send the message.

`C-x m` Begin composing a message to send (`mail`).

`C-x 4 m` Likewise, but display the message in another window (`mail-other-window`).

`C-c C-c` In Mail mode, send the message and switch to another buffer (`mail-send-and-exit`).

The command `C-x m` (`mail`) selects a buffer named '`*mail*`' and initializes it with the skeleton of an outgoing message. `C-x 4 m` (`mail-other-window`) selects the '`*mail*`' buffer in a different window, leaving the previous current buffer visible.

Because the mail composition buffer is an ordinary Emacs buffer, you can switch to other buffers while in the middle of composing mail, and switch back later (or never). If you use the `C-x m` command again when you have been composing another message but have not sent it, you are asked to confirm before the old message is erased. If you answer `n`, the '`*mail*`' buffer is left selected with its old contents, so you can finish the old message and send it. `C-u C-x m` is another way to do this. Sending the message marks the '`*mail*`' buffer "unmodified", which avoids the need for confirmation when `C-x m` is next used.

If you are composing a message in the '`*mail*`' buffer and want to send another message before finishing the first, rename the '`*mail*`' buffer using `M-x rename-buffer` (see Section 16.3 [Misc Buffer], page 107).

25.1 The Format of the Mail Buffer

In addition to the *text* or contents, a message has *header fields* which say who sent it, when, to whom, why, and so on. Some header fields such as the date and sender are created automatically after the message is sent. Others, such as the recipient names, must be specified by you in order to send the message properly.

Mail mode provides a few commands to help you edit some header fields, and some are preinitialized in the buffer automatically at times. You can insert or edit any header fields using ordinary editing commands.

The line in the buffer that says

```
--text follows this line--
```

is a special delimiter that separates the headers you have specified from the text. Whatever follows this line is the text of the message; the headers precede it. The delimiter line itself does not appear in the message actually sent. The text used for the delimiter line is controlled by the variable `mail-header-separator`.

Here is an example of what the headers and text in the '`*mail*`' buffer might look like.

```
To: rms@mc
CC: mly@mc, rg@oz
Subject: The Emacs Manual
--Text follows this line--
Please ignore this message.
```

25.2 Mail Header Fields

There are several header fields you can use in the '`*mail*`' buffer. Each header field starts with a field name at the beginning of a line, terminated by a colon. It does not matter whether you use upper or lower case in the field name. After the colon and optional whitespace comes the contents of the field.

'`To`' This field contains the mailing addresses to which the message is addressed.

'`Subject`' The contents of the '`Subject`' field should be a piece of text that says what the message is about. The reason '`Subject`' fields are useful is that most mail-reading programs can provide a summary of messages, listing the subject of each message but not its text.

'`CC`' This field contains additional mailing addresses to send the message to, but whose readers should not regard the message as addressed to them.

'`BCC`' This field contains additional mailing addresses to send the message to, but which should not appear in the header of the message actually sent.

'`FCC`' This field contains the name of one file (in Unix mail file format) to which a copy of the message should be appended when the message is sent.

'`From`' Use the '`From`' field to say who you are, when the account you are using to send the mail is not your own. The contents of the '`From`' field should be a valid mailing address, since replies will normally go there.

'Reply-To'
: Use the 'Reply-to' field to direct replies to a different address, not your own. There is no difference between 'From' and 'Reply-to' in their effect on where replies go, but they convey a different meaning to the human who reads the message.

'In-Reply-To'
: This field contains a piece of text describing a message you are replying to. Some mail systems can use this information to correlate related pieces of mail. Normally this field is filled in by Rmail when you are replying to a message in Rmail, and you never need to think about it (see Chapter 26 [Rmail], page 193).

The 'To', 'CC', 'BCC' and 'FCC' fields can appear any number of times, to specify many places to send the message.

The 'To', 'CC', and 'BCC' fields can have continuation lines. All the lines starting with whitespace, following the line on which the field starts, are considered part of the field. For example,

```
To: foo@here, this@there,
  me@gnu.cambridge.mass.usa.earth.spiral3281
```

If you have a '~/.mailrc' file, Emacs will scan it for mail aliases the first time you try to send mail in an Emacs session. Aliases found in the 'To', 'CC', and 'BCC' fields will be expanded where appropriate.

If the variable mail-archive-file-name is non-nil, it should be a string naming a file; every time you start to edit a message to send, an 'FCC' field will be put in for that file. Unless you remove the 'FCC' field, every message will be written into that file when it is sent.

25.3 Mail Mode

The major mode used in the '*mail*' buffer is Mail mode, which is much like Text mode except that various special commands are provided on the C-c prefix. These commands all have to do specifically with editing or sending the message.

C-c C-s
: Send the message, and leave the '*mail*' buffer selected (mail-send).

C-c C-c
: Send the message, and select some other buffer (mail-send-and-exit).

C-c C-f C-t
: Move to the 'To' header field, creating one if there is none (mail-to).

`C-c C-f C-s`
: Move to the '`Subject`' header field, creating one if there is none (`mail-subject`).

`C-c C-f C-c`
: Move to the '`CC`' header field, creating one if there is none (`mail-cc`).

`C-c C-w` Insert the file '`~/.signature`' at the end of the message text (`mail-signature`).

`C-c C-y` Yank the selected message from Rmail (`mail-yank-original`). This command does nothing unless your command to start sending a message was issued with Rmail.

`C-c C-q` Fill all paragraphs of yanked old messages, each individually (`mail-fill-yanked-message`).

There are two ways to send the message. `C-c C-s` (`mail-send`) sends the message and marks the '`*mail*`' buffer unmodified, but leaves that buffer selected so that you can modify the message (perhaps with new recipients) and send it again. `C-c C-c` (`mail-send-and-exit`) sends and then deletes the window (if there is another window) or switches to another buffer. It puts the '`*mail*`' buffer at the lowest priority for automatic reselection, since you are finished with using it. This is the usual way to send the message.

Mail mode provides some other special commands that are useful for editing the headers and text of the message before you send it. There are three commands defined to move point to particular header fields, all based on the prefix `C-c C-f` ('`C-f`' is for "field"). They are `C-c C-f C-t` (`mail-to`) to move to the '`To`' field, `C-c C-f C-s` (`mail-subject`) for the '`Subject`' field, and `C-c C-f C-c` (`mail-cc`) for the '`CC`' field. These fields have special motion commands because they are the most common fields for the user to want to edit.

`C-c C-w` (`mail-signature`) adds a standard piece text at the end of the message to say more about who you are. The text comes from the file '`.signature`' in your home directory.

When mail sending is invoked from the Rmail mail reader using an Rmail command, `C-c C-y` can be used inside the '`*mail*`' buffer to insert the text of the message you are replying to. Normally it indents each line of that message four spaces and eliminates most header fields. A numeric argument specifies the number of spaces to indent. An argument of just `C-u` says not to indent at all and not to eliminate anything. `C-c C-y` always uses the current message from the '`RMAIL`' buffer, so you can insert several old messages by selecting one in '`RMAIL`', switching to '`*mail*`' and yanking it, then switching back to '`RMAIL`' to select another.

After using `C-c C-y`, the command `C-c C-q` (`mail-fill-yanked-message`) can be used to fill the paragraphs of the yanked old message or messages. One use of `C-c C-q` fills all such paragraphs, each one separately.

Turning on Mail mode (which `C-x m` does automatically) calls the value of `text-mode-hook`, if it is not void or `nil`, and then calls the value of `mail-mode-hook` if that is not void or `nil`.

26 Reading Mail with Rmail

Rmail is an Emacs subsystem for reading and disposing of mail that you receive. Rmail stores mail messages in files called Rmail files. Reading the message in an Rmail file is done in a special major mode, Rmail mode, which redefines most letters to run commands for managing mail. To enter Rmail, type `M-x rmail`. This reads your primary mail file, merges new mail in from your inboxes, displays the first new message, and lets you begin reading.

Using Rmail in the simplest fashion, you have one Rmail file '`~/RMAIL`' in which all of your mail is saved. It is called your *primary mail file*. In more sophisticated usage, you can copy messages into other Rmail files and then edit those files with Rmail.

Rmail displays only one message at a time. It is called the *current message*. Rmail mode's special commands can do such things as move to another message, delete the message, copy the message into another file, or send a reply.

Within the Rmail file, messages are arranged sequentially in order of receipt. They are also assigned consecutive integers as their *message numbers*. The number of the current message is displayed in Rmail's mode line, followed by the total number of messages in the file. You can move to a message by specifying its message number using the `j` key (see Section 26.2 [Rmail Motion], page 194).

Following the usual conventions of Emacs, changes in an Rmail file become permanent only when the file is saved. You can do this with `s` (`rmail-save`), which also expunges deleted messages from the file first (see Section 26.3 [Rmail Deletion], page 195). To save the file without expunging, use `C-x C-s`. Rmail saves the Rmail file spontaneously when moving new mail from an inbox file (see Section 26.4 [Rmail Inbox], page 197).

You can exit Rmail with `q` (`rmail-quit`); this expunges and saves the Rmail file and then switches to another buffer. But there is no need to 'exit' formally. If you switch from Rmail to editing in other buffers, and never happen to switch back, you have exited. Just make sure to save the Rmail file eventually (like any other file you have changed). `C-x s` is a good enough way to do this (see Section 15.3 [Saving], page 90).

26.1 Scrolling Within a Message

When Rmail displays a message that does not fit on the screen, it is necessary to scroll through it. This could be done with `C-v`, `M-v` and `M-<`, but in Rmail scrolling is so frequent that it deserves to be easier to type.

`SPC` Scroll forward (`scroll-up`).

`DEL` Scroll backward (`scroll-down`).

`.` Scroll to start of message (`rmail-beginning-of-message`).

Since the most common thing to do while reading a message is to scroll through it by screenfuls, Rmail makes `SPC` and `DEL` synonyms of `C-v` (`scroll-up`) and `M-v` (`scroll-down`)

The command `.` (`rmail-beginning-of-message`) scrolls back to the beginning of the selected message. This is not quite the same as `M-<`: for one thing, it does not set the mark; for another, it resets the buffer boundaries to the current message if you have changed them.

26.2 Moving Among Messages

The most basic thing to do with a message is to read it. The way to do this in Rmail is to make the message current. You can make any message current given its message number using the `j` command, but the usual thing to do is to move sequentially through the file, since this is the order of receipt of messages. When you enter Rmail, you are positioned at the first new message (new messages are those received since the previous use of Rmail), or at the last message if there are no new messages this time. Move forward to see the other new messages; move backward to reexamine old messages.

`n` Move to the next nondeleted message, skipping any intervening deleted messages (`rmail-next-undeleted-message`).

`p` Move to the previous nondeleted message (`rmail-previous-undeleted-message`).

`M-n` Move to the next message, including deleted messages (`rmail-next-message`).

`M-p` Move to the previous message, including deleted messages (`rmail-previous-message`).

`j` Move to the first message. With argument *n*, move to message number *n* (`rmail-show-message`).

`>` Move to the last message (`rmail-last-message`).

`M-s` *regexp* `RET`
Move to the next message containing a match for *regexp* (`rmail-search`). If *regexp* is empty, the last regexp used is used again.

`- M-s` *regexp* `RET`
: Move to the previous message containing a match for *regexp*. If *regexp* is empty, the last regexp used is used again.

`n` and `p` are the usual way of moving among messages in Rmail. They move through the messages sequentially, but skip over deleted messages, which is usually what you want to do. Their command definitions are named `rmail-next-undeleted-message` and `rmail-previous-undeleted-message`. If you do not want to skip deleted messages—for example, if you want to move to a message to undelete it—use the variants `M-n` and `M-p` (`rmail-next-message` and `rmail-previous-message`). A numeric argument to any of these commands serves as a repeat count.

In Rmail, you can specify a numeric argument by typing the digits. It is not necessary to type `C-u` first.

The `M-s` (`rmail-search`) command is Rmail's version of search. The usual incremental search command `C-s` works in Rmail, but it searches only within the current message. The purpose of `M-s` is to search for another message. It reads a regular expression (see Section 13.5 [Regexps], page 74) nonincrementally, then searches starting at the beginning of the following message for a match. The message containing the match is selected.

To search backward in the file for another message, give `M-s` a negative argument. In Rmail this can be done with `- M-s`.

It is also possible to search for a message based on labels. See Section 26.7 [Rmail Labels], page 199.

To move to a message specified by absolute message number, use `j` (`rmail-show-message`) with the message number as argument. With no argument, `j` selects the first message. `>` (`rmail-last-message`) selects the last message.

26.3 Deleting Messages

When you no longer need to keep a message, you can *delete* it. This flags it as ignorable, and some Rmail commands will pretend it is no longer present; but it still has its place in the Rmail file, and still has its message number.

Expunging the Rmail file actually removes the deleted messages. The remaining messages are renumbered consecutively. Expunging is the only action that changes the message number of any message, except for undigestifying (see Chapter 27 [Rmail Digest], page 207).

`d` Delete the current message, and move to the next nondeleted message (`rmail-delete-forward`).

`C-d` Delete the current message, and move to the previous nondeleted message (`rmail-delete-backward`).

`u` Undelete the current message, or move back to a deleted message and undelete it (`rmail-undelete-previous-message`).

`x`

`e` Expunge the Rmail file (`rmail-expunge`). These two commands are synonyms.

There are two Rmail commands for deleting messages. Both delete the current message and select another message. `d` (`rmail-delete-forward`) moves to the following message, skipping messages already deleted, while `C-d` (`rmail-delete-backward`) moves to the previous nondeleted message. If there is no nondeleted message to move to in the specified direction, the message that was just deleted remains current.

To make all the deleted messages finally vanish from the Rmail file, type `e` (`rmail-expunge`). Until you do this, you can still *undelete* the deleted messages.

To undelete, type `u` (`rmail-undelete-previous-message`), which is designed to cancel the effect of a `d` command (usually). It undeletes the current message if the current message is deleted. Otherwise it moves backward to previous messages until a deleted message is found, and undeletes that message.

You can usually undo a `d` with a `u` because the `u` moves back to and undeletes the message that the `d` deleted. But this does not work when the `d` skips a few already-deleted messages that follow the message being deleted; then the `u` command will undelete the last of the messages that were skipped. There is no clean way to avoid this problem. However, by repeating the `u` command, you can eventually get back to the message that you intended to undelete. You can also reach that message with `M-p` commands and then type `u`.

A deleted message has the '`deleted`' attribute, and as a result '`deleted`' appears in the mode line when the current message is deleted. In fact, deleting or undeleting a message is nothing more than adding or removing this attribute. See Section 26.7 [Rmail Labels], page 199.

26.4 Rmail Files and Inboxes

Unix places incoming mail for you in a file that we call your *inbox*. When you start up Rmail, it copies the new messages from your inbox into your primary mail file, an Rmail file, which also contains other messages saved from previous Rmail sessions. It is in this file that you actually read the mail with Rmail. This operation is called *getting new mail*. It can be repeated at any time using the `g` key in Rmail. The inbox file name is '`/usr/spool/mail/`*username*' in Berkeley Unix, '`/usr/mail/`*username*' in system V.

There are two reason for having separate Rmail files and inboxes.

1. The format in which Unix delivers the mail in the inbox is not adequate for Rmail mail storage. It has no way to record attributes (such as '`deleted`') or user-specified labels; it has no way to record old headers and reformatted headers; it has no way to record cached summary line information.
2. It is very cumbersome to access an inbox file without danger of losing mail, because it is necessary to interlock with mail delivery. Moreover, different Unix systems use different interlocking techniques. The strategy of moving mail out of the inbox once and for all into a separate Rmail file avoids the need for interlocking in all the rest of Rmail, since only Rmail operates on the Rmail file.

When getting new mail, Rmail first copies the new mail from the inbox file to the Rmail file; then it saves the Rmail file; then it deletes the inbox file. This way, a system crash may cause duplication of mail between the inbox and the Rmail file, but cannot lose mail.

Copying mail from an inbox in the system's mailer directory actually puts it in an intermediate file '`~/.newmail`'. This is because the interlocking is done by a C program that copies to another file. '`~/.newmail`' is deleted after mail merging is successful. If there is a crash at the wrong time, this file will continue to exist and will be used as an inbox the next time you get new mail.

26.5 Multiple Mail Files

Rmail operates by default on your *primary mail file*, which is named '`~/RMAIL`' and receives your incoming mail from your system inbox file. But you can also have other mail files and edit them with Rmail. These files can receive mail through their own inboxes, or you can move messages into them by explicit command in Rmail (see Section 26.6 [Rmail Output], page 198).

`i` *file* `RET` Read *file* into Emacs and run Rmail on it (`rmail-input`).

`M-x set-rmail-inbox-list RET` *files* `RET`
: Specify inbox file names for current Rmail file to get mail from.

`g`
: Merge new mail from current Rmail file's inboxes (`rmail-get-new-mail`).

`C-u g` *file*
: Merge new mail from inbox file *file*.

To run Rmail on a file other than your primary mail file, you may use the `i` (`rmail-input`) command in Rmail. This visits the file, puts it in Rmail mode, and then gets new mail from the file's inboxes if any. You can also use `M-x rmail-input` even when not in Rmail.

The file you read with `i` does not have to be in Rmail file format. It could also be Unix mail format, or mmdf format; or it could be a mixture of all three, as long as each message belongs to one of the three formats. Rmail recognizes all three and converts all the messages to proper Rmail format before showing you the file.

Each Rmail file can contain a list of inbox file names; you can specify this list with `M-x set-rmail-inbox-list RET` *files* `RET`. The argument can contain any number of file names, separated by commas. It can also be empty, which specifies that this file should have no inboxes. Once a list of inboxes is specified, the Rmail file remembers it permanently until it is explicitly changed.

If an Rmail file has inboxes, new mail is merged in from the inboxes when the Rmail file is brought into Rmail, and when the `g` (`rmail-get-new-mail`) command is used. If the Rmail file specifies no inboxes, then no new mail is merged in at these times. A special exception is made for your primary mail file in using the standard system inbox for it if it does not specify any.

To merge mail from a file that is not the usual inbox, give the `g` key a numeric argument, as in `C-u g`. Then it reads a file name and merges mail from that file. The inbox file is not deleted or changed in any way when `g` with an argument is used. This is, therefore, a general way of merging one file of messages into another.

26.6 Copying Messages Out to Files

`o` *file* `RET`
: Append a copy of the current message to the file *file*, writing it in Rmail file format (`rmail-output-to-rmail-file`).

`C-o` *file* `RET`
: Append a copy of the current message to the file *file*, writing it in Unix mail file format (`rmail-output`).

If an Rmail file has no inboxes, how does it get anything in it? By explicit `o` commands.

`o` (`rmail-output-to-rmail-file`) appends the current message in Rmail format to the end of the specified file. This is the best command to use to move messages between Rmail files. If the other Rmail file is currently visited, the copying is done into the other file's Emacs buffer instead. You should eventually save it on disk.

The `C-o` (`rmail-output`) command in Rmail appends a copy of the current message to a specified file, in Unix mail file format. This is useful for moving messages into files to be read by other mail processors that do not understand Rmail format.

Copying a message with `o` or `C-o` gives the original copy of the message the '`filed`' attribute, so that '`filed`' appears in the mode line when such a message is current.

Normally you should use only `o` to output messages to other Rmail files, never `C-o`. But it is also safe if you always use `C-o`, never `o`. When a file is visited in Rmail, the last message is checked, and if it is in Unix format, the entire file is scanned and all Unix-format messages are converted to Rmail format. (The reason for checking the last message is that scanning the file is slow and most Rmail files have only Rmail format messages.) If you use `C-o` consistently, the last message is sure to be in Unix format, so Rmail will convert all messages properly.

The case where you might want to use `C-o` always, instead of `o` always, is when you or other users want to append mail to the same file from other mail processors. Other mail processors probably do not know Rmail format but do know Unix format.

In any case, always use `o` to add to an Rmail file that is being visited in Rmail. Adding messages with `C-o` to the actual disk file will trigger a "simultaneous editing" warning when you ask to save the Emacs buffer, and will be lost if you do save.

26.7 Labels

Each message can have various *labels* assigned to it as a means of classification. A label has a name; different names mean different labels. Any given label is either present or absent on a particular message. A few label names have standard meanings and are given to messages automatically by Rmail when appropriate; these special labels are called *attributes*. All other labels are assigned by the user.

`a` *label* `RET`

Assign the label *label* to the current message (`rmail-add-label`).

`k` *label* `RET`
: Remove the label *label* from the current message (`rmail-kill-label`).

`C-M-n` *labels* `RET`
: Move to the next message that has one of the labels *labels* (`rmail-next-labeled-message`).

`C-M-p` *labels* `RET`
: Move to the previous message that has one of the labels *labels* (`rmail-previous-labeled-message`).

`C-M-l` *labels* `RET`
: Make a summary of all messages containing any of the labels *labels* (`rmail-summary-by-labels`).

Specifying an empty string for one these commands means to use the last label specified for any of these commands.

The `a` (`rmail-add-label`) and `k` (`rmail-kill-label`) commands allow you to assign or remove any label on the current message. If the *label* argument is empty, it means to assign or remove the same label most recently assigned or removed.

Once you have given messages labels to classify them as you wish, there are two ways to use the labels: in moving and in summaries.

The command `C-M-n` *labels* `RET` (`rmail-next-labeled-message`) moves to the next message that has one of the labels *labels*. *labels* is one or more label names, separated by commas. `C-M-p` (`rmail-previous-labeled-message`) is similar, but moves backwards to previous messages. A preceding numeric argument to either one serves as a repeat count.

The command `C-M-l` *labels* `RET` (`rmail-summary-by-labels`) displays a summary containing only the messages that have at least one of a specified set of messages. The argument *labels* is one or more label names, separated by commas. See Section 26.8 [Rmail Summary], page 201, for information on summaries.

If the *labels* argument to `C-M-n`, `C-M-p` or `C-M-l` is empty, it means to use the last set of labels specified for any of these commands.

Some labels such as '`deleted`' and '`filed`' have built-in meanings and are assigned to or removed from messages automatically at appropriate times; these labels are called *attributes*. Here is a list of Rmail attributes:

'unseen' Means the message has never been current. Assigned to messages when they come from an inbox file, and removed when a message is made current.

'deleted' Means the message is deleted. Assigned by deletion commands and removed by undeletion commands (see Section 26.3 [Rmail Deletion], page 195).

'filed' Means the message has been copied to some other file. Assigned by the file output commands (see Section 26.5 [Rmail Files], page 197).

'answered'
Means you have mailed an answer to the message. Assigned by the `r` command (`rmail-reply`). See Section 26.9 [Rmail Reply], page 204.

'forwarded'
Means you have forwarded the message to other users. Assigned by the `f` command (`rmail-forward`). See Section 26.9 [Rmail Reply], page 204.

'edited' Means you have edited the text of the message within Rmail. See Section 26.10 [Rmail Editing], page 205.

All other labels are assigned or removed only by the user, and it is up to the user to decide what they mean.

26.8 Summaries

A *summary* is a buffer containing one line per message that Rmail can make and display to give you an overview of the mail in an Rmail file. Each line shows the message number, the sender, the labels, and the subject. When the summary buffer is selected, various commands can be used to select messages by moving in the summary buffer, or delete or undelete messages.

A summary buffer applies to a single Rmail file only; if you are editing multiple Rmail files, they have separate summary buffers. The summary buffer name is made by appending '`-summary`' to the Rmail buffer's name. Only one summary buffer will be displayed at a time unless you make several windows and select the summary buffers by hand.

26.8.1 Making Summaries

Here are the commands to create a summary for the current Rmail file. Summaries do not update automatically; to make an updated summary, you must use one of these commands again.

h

`C-M-h` Summarize all messages (`rmail-summary`).

`l` *labels* `RET`
`C-M-l` *labels* `RET`
: Summarize message that have one or more of the specified labels (`rmail-summary-by-labels`).

`C-M-r` *rcpts* `RET`
: Summarize messages that have one or more of the specified recipients (`rmail-summary-by-recipients`)

The `h` or `C-M-h` (`rmail-summary`) command fills the summary buffer for the current Rmail file with a summary of all the messages in the file. It then displays and selects the summary buffer in another window.

`C-M-l` *labels* `RET` (`rmail-summary-by-labels`) makes a partial summary mentioning only the messages that have one or more of the labels *labels*. *labels* should contain label names separated by commas.

`C-M-r` *rcpts* `RET` (`rmail-summary-by-recipients`) makes a partial summary mentioning only the messages that have one or more of the recipients *rcpts*. *rcpts* should contain mailing addresses separated by commas.

Note that there is only one summary buffer for any Rmail file; making one kind of summary discards any previously made summary.

26.8.2 Editing in Summaries

Summary buffers are given the major mode Rmail Summary mode, which provides the following special commands:

`j` Select the message described by the line that point is on (`rmail-summary-goto-msg`).

`C-n` Move to next line and select its message in Rmail (`rmail-summary-next-all`).

`C-p` Move to previous line and select its message (`rmail-summary-previous-all`).

`n` Move to next line, skipping lines saying 'deleted', and select its message (`rmail-summary-next-msg`).

`p` Move to previous line, skipping lines saying 'deleted', and select its message (`rmail-summary-previous-msg`).

`d` Delete the current line's message, then do like `n` (`rmail-summary-delete-forward`).

u Undelete and select this message or the previous deleted message in the summary (rmail-summary-undelete).

SPC Scroll the other window (presumably Rmail) forward (rmail-summary-scroll-msg-up).

DEL Scroll the other window backward (rmail-summary-scroll-msg-down).

x Kill the summary window (rmail-summary-exit).

q Exit Rmail (rmail-summary-quit).

The keys C-n and C-p are modified in Rmail Summary mode so that in addition to moving point in the summary buffer they also cause the line's message to become current in the associated Rmail buffer. That buffer is also made visible in another window if it is not already so.

n and p are similar to C-n and C-p, but skip lines that say 'message deleted'. They are like the n and p keys of Rmail itself. Note, however, that in a partial summary these commands move only among the message listed in the summary.

The other Emacs cursor motion commands are not changed in Rmail Summary mode, so it is easy to get the point on a line whose message is not selected in Rmail. This can also happen if you switch to the Rmail window and switch messages there. To get the Rmail buffer back in sync with the summary, use the j (rmail-summary-goto-msg) command, which selects in Rmail the message of the current summary line.

Deletion and undeletion can also be done from the summary buffer. They always work based on where point is located in the summary buffer, ignoring which message is selected in Rmail. d (rmail-summary-delete-forward) deletes the current line's message, then moves to the next line whose message is not deleted and selects that message. The inverse of this is u (rmail-summary-undelete), which moves back (if necessary) to a line whose message is deleted, undeletes that message, and selects it in Rmail.

When moving through messages with the summary buffer, it is convenient to be able to scroll the message while remaining in the summary window. The commands SPC (rmail-summary-scroll-msg-up) and DEL (rmail-summary-scroll-msg-down) do this. They scroll the message just as those same keys do when the Rmail buffer is selected.

When you are finished using the summary, type x (rmail-summary-exit) to kill the summary buffer's window.

You can also exit Rmail while in the summary. q (rmail-summary-quit) kills the summary window, then saves the Rmail file and switches to another buffer.

26.9 Sending Replies

Rmail has several commands that use Mail mode to send outgoing mail. See Chapter 25 [Sending Mail], page 187, for information on using Mail mode. What are documented here are the special commands of Rmail for entering Mail mode. Note that the usual keys for sending mail, `C-x m` and `C-x 4 m`, are available in Rmail mode and work just as they usually do.

`m` Send a message (`rmail-mail`).

`c` Continue editing already started outgoing message (`rmail-continue`).

`r` Send a reply to the current Rmail message (`rmail-reply`).

`f` Forward current message to other users (`rmail-forward`).

The most common reason to send a message while in Rmail is to reply to the message you are reading. To do this, type `r` (`rmail-reply`). This displays the '`*mail*`' buffer in another window, much like `C-x 4 m`, but preinitializes the '`Subject`', '`To`', '`CC`' and '`In-reply-to`' header fields based on the message being replied to. The '`To`' field is given the sender of that message, and the '`CC`' gets all the recipients of that message (but recipients that match elements of the list `rmail-dont-reply-to` are omitted; by default, this list contains your own mailing address).

Once you have initialized the '`*mail*`' buffer this way, sending the mail goes as usual (see Chapter 25 [Sending Mail], page 187). You can edit the presupplied header fields if they are not right for you.

One additional Mail mode command is available when mailing is invoked from Rmail: `C-c C-y` (`mail-yank-original`) inserts into the outgoing message a copy of the current Rmail message; normally this is the message you are replying to, but you can also switch to the Rmail buffer, select a different message, switch back, and yank new current message. Normally the yanked message is indented four spaces and has most header fields deleted from it; an argument to `C-c C-y` specifies the amount to indent, and `C-u C-c C-y` does not indent at all and does not delete any header fields.

Another frequent reason to send mail in Rmail is to forward the current message to other users. `f` (`rmail-forward`) makes this easy by preinitializing the '`*mail*`' buffer with the current message as the text, and a subject designating a forwarded message. All you have to do is fill in the recipients and send.

The `m` (`rmail-mail`) command is used to start editing an outgoing message that is not a reply. It leaves the header fields empty. Its only difference from `C-x 4 m` is that it makes the Rmail buffer accessible for `C-c y`, just as `r` does. Thus, `m` can be used to reply to or forward a message; it can do anything `r` or `f` can do.

The `c` (`rmail-continue`) command resumes editing the '`*mail*`' buffer, to finish editing an outgoing message you were already composing, or to alter a message you have sent.

26.10 Editing Within a Message

Rmail mode provides a few special commands for moving within and editing the current message. In addition, the usual Emacs commands are available (except for a few, such as `C-M-n` and `C-M-h`, that are redefined by Rmail for other purposes). However, the Rmail buffer is normally read-only, and to alter it you must use the Rmail command `w` described below.

`t` Toggle display of original headers (`rmail-toggle-headers`).

`w` Edit current message (`rmail-edit-current-message`).

Rmail reformats the header of each message before displaying it. Normally this involves deleting most header fields, on the grounds that they are not interesting. The variable `rmail-ignored-headers` should contain a regexp that matches the header fields to discard in this way. The original headers are saved permanently, and to see what they look like, use the `t` (`rmail-toggle-headers`) command. This discards the reformatted headers of the current message and displays it with the original headers. Repeating `t` reformats the message again. Selecting the message again also reformats.

The Rmail buffer is normally read only, and most of the characters you would type to modify it (including most letters) are redefined as Rmail commands. This is usually not a problem since it is rare to want to change the text of a message. When you do want to do this, the way is to type `w` (`rmail-edit-current-message`), which changes from Rmail mode into Rmail Edit mode, another major mode which is nearly the same as Text mode. The mode line illustrates this change.

In Rmail Edit mode, letters insert themselves as usual and the Rmail commands are not available. When you are finished editing the message and are ready to go back to Rmail, type `C-c C-c`, which switches back to Rmail mode. Alternatively, you can return to Rmail mode but cancel all the editing that you have done by typing `C-c C-]`.

Entering Rmail Edit mode calls with no arguments the value of the variable `text-mode-hook`, if that value exists and is not `nil`; then it does the same with the variable `rmail-edit-mode-hook`. It adds the attribute '`edited`' to the message.

26.11 Digest Messages

A *digest message* is a message which exists to contain and carry several other messages. Digests are used on moderated mailing lists; all the messages that arrive for the list during a period of time such as one day are put inside a single digest which is then sent to the subscribers. Transmitting the single digest uses much less computer time than transmitting the individual messages even though the total size is the same, because the per-message overhead in network mail transmission is considerable.

When you receive a digest message, the most convenient way to read it is to *undigestify* it: to turn it back into many individual messages. Then you can read and delete the individual messages as it suits you.

To undigestify a message, select it and then type `M-x undigestify-rmail-message`. This copies each submessage as a separate Rmail message and inserts them all following the digest. The digest message itself is flagged as deleted.

27 Miscellaneous Commands

This chapter contains several brief topics that do not fit anywhere else.

27.1 Recursive Editing Levels

A *recursive edit* is a situation in which you are using Emacs commands to perform arbitrary editing while in the middle of another Emacs command. For example, when you type `C-r` inside of a `query-replace`, you enter a recursive edit in which you can change the current buffer. On exiting from the recursive edit, you go back to the `query-replace`.

Exiting the recursive edit means returning to the unfinished command, which continues execution. For example, exiting the recursive edit requested by `C-r` in `query-replace` causes query replacing to resume. Exiting is done with `C-M-c` (`exit-recursive-edit`).

You can also *abort* the recursive edit. This is like exiting, but also quits the unfinished command immediately. Use the command `C-]` (`abort-recursive-edit`) for this. See Section 29.1 [Quitting], page 237.

The mode line shows you when you are in a recursive edit by displaying square brackets around the parentheses that always surround the major and minor mode names. Every window's mode line shows this, in the same way, since being in a recursive edit is true of Emacs as a whole rather than any particular buffer.

It is possible to be in recursive edits within recursive edits. For example, after typing `C-r` in a `query-replace`, you might type a command that entered the debugger. In such circumstances, two or more sets of square brackets appear in the mode line. Exiting the inner recursive edit (such as, with the debugger `c` command) would resume the command where it called the debugger. After the end of this command, you would be able to exit the first recursive edit. Aborting also gets out of only one level of recursive edit; it returns immediately to the command level of the previous recursive edit. So you could immediately abort that one too.

Alternatively, the command `M-x top-level` aborts all levels of recursive edits, returning immediately to the top level command reader.

The text being edited inside the recursive edit need not be the same text that you were editing at top level. It depends on what the recursive edit is for. If the command that invokes the recursive edit selects a different buffer first, that is the buffer you will edit recursively. In any case, you can switch buffers within the recursive edit in the normal manner (as long as the buffer-switching keys have not been rebound). You could probably do all the rest of your editing inside the recursive edit, visiting files and all. But this could have surprising effects (such as stack overflow) from time to time. So remember to exit or abort the recursive edit when you no longer need it.

In general, GNU Emacs tries to avoid using recursive edits. It is usually preferable to allow the user to switch among the possible editing modes in any order he likes. With recursive edits, the only way to get to another state is to go "back" to the state that the recursive edit was invoked from.

27.2 Narrowing

Narrowing means focusing in on some portion of the buffer, making the rest temporarily invisible and inaccessible. Cancelling the narrowing, and making the entire buffer once again visible, is called *widening*. The amount of narrowing in effect in a buffer at any time is called the buffer's *restriction*.

`C-x n` Narrow down to between point and mark (`narrow-to-region`).

`C-x w` Widen to make the entire buffer visible again (`widen`).

When you have narrowed down to a part of the buffer, that part appears to be all there is. You can't see the rest, you can't move into it (motion commands won't go outside the visible part), you can't change it in any way. However, it is not gone, and if you save the file all the invisible text will be saved. In addition to sometimes making it easier to concentrate on a single subroutine or paragraph by eliminating clutter, narrowing can be used to restrict the range of operation of a replace command or repeating keyboard macro. The word '`Narrow`' appears in the mode line whenever narrowing is in effect.

The primary narrowing command is `C-x n` (`narrow-to-region`). It sets the current buffer's restrictions so that the text in the current region remains visible but all text before the region or after the region is invisible. Point and mark do not change.

Because narrowing can easily confuse users who do not understand it, `narrow-to-region` is normally a disabled command. Attempting to use this command asks for confirmation and gives you the option of enabling it; once you enable the command, confirmation will no longer be required for it. See Section 28.4.3 [Disabling], page 229.

The way to undo narrowing is to widen with `C-x w` (`widen`). This makes all text in the buffer accessible again.

You can get information on what part of the buffer you are narrowed down to using the `C-x =` command. See Section 4.8 [Position Info], page 30.

27.3 Sorting Text

Emacs provides several commands for sorting text in the buffer. All operate on the contents of the region (the text between point and the mark). They divide the text of the region into many *sort records*, identify a *sort key* for each record, and then reorder the records into the order determined by the sort keys. The records are ordered so that their keys are in alphabetical order, or, for numeric sorting, in numeric order. In alphabetic sorting, all upper case letters 'A' through 'Z' come before lower case 'a', in accord with the ASCII character sequence.

The various sort commands differ in how they divide the text into sort records and in which part of each record is used as the sort key. Most of the commands make each line a separate sort record, but some commands use paragraphs or pages as sort records. Most of the sort commands use each entire sort record as its own sort key, but some use only a portion of the record as the sort key.

`M-x sort-lines`
: Divide the region into lines, and sort by comparing the entire text of a line. A prefix argument means sort into descending order.

`M-x sort-paragraphs`
: Divide the region into paragraphs, and sort by comparing the entire text of a paragraph (except for leading blank lines). A prefix argument means sort into descending order.

`M-x sort-pages`
: Divide the region into pages, and sort by comparing the entire text of a page (except for leading blank lines). A prefix argument means sort into descending order.

`M-x sort-fields`
: Divide the region into lines, and sort by comparing the contents of one field in each line. Fields are defined as separated by whitespace, so the first run of consecutive non-whitespace characters in a line constitutes field 1, the second such run constitutes field 2, etc.

  You specify which field to sort by with a numeric argument: 1 to sort by field 1, etc. A negative argument means sort into descending order. Thus, minus 2 means sort by field 2 in reverse-alphabetical order.

`M-x sort-numeric-fields`
: Like `M-x sort-fields` except the specified field is converted to a number for each line, and the numbers are compared. '10' comes before '2' when considered as text, but after it when considered as a number.

`M-x sort-columns`
: Like `M-x sort-fields` except that the text within each line used for comparison comes from a fixed range of columns. See below for an explanation.

For example, if the buffer contains

```
On systems where clash detection (locking of files being edited) is
implemented, Emacs also checks the first time you modify a buffer
whether the file has changed on disk since it was last visited or
saved.  If it has, you are asked to confirm that you want to change
the buffer.
```

then if you apply `M-x sort-lines` to the entire buffer you get

```
On systems where clash detection (locking of files being edited) is
implemented, Emacs also checks the first time you modify a buffer
saved.  If it has, you are asked to confirm that you want to change
the buffer.
whether the file has changed on disk since it was last visited or
```

where the upper case 'O' comes before all lower case letters. If you apply instead `C-u 2 M-x sort-fields` you get

```
implemented, Emacs also checks the first time you modify a buffer
saved.  If it has, you are asked to confirm that you want to change
the buffer.
On systems where clash detection (locking of files being edited) is
whether the file has changed on disk since it was last visited or
```

where the sort keys were '`Emacs`', '`If`', '`buffer`', '`systems`' and '`the`'.

`M-x sort-columns` requires more explanation. You specify the columns by putting point at one of the columns and the mark at the other column. Because this means you cannot put point or the mark at the beginning of the first line to sort, this command uses an unusual definition of 'region': all of the line point is in is considered part of the region, and so is all of the line the mark is in.

For example, to sort a table by information found in columns 10 to 15, you could put the mark on column 10 in the first line of the table, and point on column 15 in the last line of the table, and then use this command. Or you could put the mark on column 15 in the first line and point on column 10 in the last line.

This can be thought of as sorting the rectangle specified by point and the mark, except that the text on each line to the left or right of the rectangle moves along with the text inside the rectangle. See Section 10.4 [Rectangles], page 59.

27.4 Running Shell Commands from Emacs

Emacs has commands for passing single command lines to inferior shell processes; it can also run a shell interactively with input and output to an Emacs buffer '`*shell*`'.

`M-!` Run a specified shell command line and display the output (`shell-command`).

`M-|` Run a specified shell command line with region contents as input; optionally replace the region with the output (`shell-command-on-region`).

`M-x shell` Run a subshell with input and output through an Emacs buffer. You can then give commands interactively.

27.4.1 Single Shell Commands

`M-!` (`shell-command`) reads a line of text using the minibuffer and creates an inferior shell to execute the line as a command. Standard input from the command comes from the null device. If the shell command produces any output, the output goes into an Emacs buffer named '`*Shell Command Output*`', which is displayed in another window but not selected. A numeric argument, as in `M-1 M-!`, directs this command to insert any output into the current buffer. In that case, point is left before the output and the mark is set after the output.

`M-|` (`shell-command-on-region`) is like `M-!` but passes the contents of the region as input to the shell command, instead of no input. If a numeric argument is used, meaning insert output in the current buffer, then the old region is deleted first and the output replaces it as the contents of the region.

Both `M-!` and `M-|` use `shell-file-name` to specify the shell to use. This variable is initialized based on your `SHELL` environment variable when Emacs is started. If the file name does not specify a directory, the directories in the list `exec-path` are searched; this list is initialized based on the environment variable PATH when Emacs is started. Your '.emacs' file can override either or both of these default initializations.

With M-! and M-|, Emacs has to wait until the shell command completes. You can quit with C-g; that terminates the shell command.

27.4.2 Interactive Inferior Shell

To run a subshell interactively, putting its typescript in an Emacs buffer, use M-x shell. This creates (or reuses) a buffer named '*shell*' and runs a subshell with input coming from and output going to that buffer. That is to say, any "terminal output" from the subshell will go into the buffer, advancing point, and any "terminal input" for the subshell comes from text in the buffer. To give input to the subshell, go to the end of the buffer and type the input, terminated by RET.

Emacs does not wait for the subshell to do anything. You can switch windows or buffers and edit them while the shell is waiting, or while it is running a command. Output from the subshell waits until Emacs has time to process it; this happens whenever Emacs is waiting for keyboard input or for time to elapse.

If you would like multiple subshells, change the name of buffer '*shell*' to something different by using M-x rename-buffer. The next use of M-x shell will create a new buffer '*shell*' with its own subshell. By renaming this buffer as well you can create a third one, and so on. All the subshells run independently and in parallel.

The file name used to load the subshell is the value of the variable explicit-shell-file-name, if that is non-nil. Otherwise, the environment variable ESHELL is used, or the environment variable SHELL if there is no ESHELL. If the file name specified is relative, the directories in the list exec-path are searched (see Section 27.4.1 [Single Shell], page 211).

As soon as the subshell is started, it is sent as input the contents of the file '~/.emacs_*shellname*', if that file exists, where *shellname* is the name of the file that the shell was loaded from. For example, if you use csh, the file sent to it is '~/.emacs_csh'.

cd, pushd and popd commands given to the inferior shell are watched by Emacs so it can keep the '*shell*' buffer's default directory the same as the shell's working directory. These commands are recognized syntactically by examining lines of input that are sent. If you use aliases for these commands, you can tell Emacs to recognize them also. For example, if the value of the variable shell-pushd-regexp matches the beginning of a shell command line, that line is regarded as a `pushd` command. Change this variable when you add aliases for '`pushd`'. Likewise, `shell-popd-regexp` and `shell-cd-regexp` are used to recognize commands with the meaning of '`popd`' and '`cd`'. These commands are recognized only at the beginning of a shell command line.

If Emacs gets an error while trying to handle what it believes is a '`cd`', '`pushd`' or '`popd`' command, and the value of `shell-set-directory-error-hook` is non-`nil`, that value is called as a function with no arguments.

27.4.3 Shell Mode

The shell buffer uses Shell mode, which defines several special keys attached to the `C-c` prefix. They are chosen to resemble the usual editing and job control characters present in shells that are not under Emacs, except that you must type `C-c` first. Here is a complete list of the special key bindings of Shell mode:

`RET` At end of buffer send line as input; otherwise, copy current line to end of buffer and send it (`send-shell-input`). When a line is copied, any text at the beginning of the line that matches the variable `shell-prompt-pattern` is left out; this variable's value should be a regexp string that matches the prompts that you use in your subshell.

`C-c C-d` Send end-of-file as input, probably causing the shell or its current subjob to finish (`shell-send-eof`).

`C-c C-u` Kill all text that has yet to be sent as input (`kill-shell-input`).

`C-c C-w` Kill a word before point (`backward-kill-word`).

`C-c C-c` Interrupt the shell or its current subjob if any (`interrupt-shell-subjob`).

`C-c C-z` Stop the shell or its current subjob if any (`stop-shell-subjob`).

`C-c C-\` Send quit signal to the shell or its current subjob if any (`quit-shell-subjob`).

`C-c C-o` Delete last batch of output from shell (`kill-output-from-shell`).

`C-c C-r` Scroll top of last batch of output to top of window (`show-output-from-shell`).

`C-c C-y` Copy the previous bunch of shell input, and insert it into the buffer before point (`copy-last-shell-input`). No final newline is inserted, and the input copied is not resubmitted until you type `RET`.

27.5 Hardcopy Output

The Emacs commands for making hardcopy derive their names from the Unix commands '`print`' and '`lpr`'.

`M-x print-buffer`
: Print hardcopy of current buffer using Unix command '`print`' ('`lpr -p`'). This makes page headings containing the file name and page number.

`M-x lpr-buffer`
: Print hardcopy of current buffer using Unix command '`lpr`'. This makes no page headings.

`M-x print-region`
: Like `print-buffer` but prints only the current region.

`M-x lpr-region`
: Like `lpr-buffer` but prints only the current region.

All the hardcopy commands pass extra switches to the `lpr` program based on the value of the variable `lpr-switches`. Its value should be a list of strings, each string a switch starting with '`-`'. For example, the value could be `("-Pfoo")` to print on printer '`foo`'.

27.6 Dissociated Press

`M-x dissociated-press` is a command for scrambling a file of text either word by word or character by character. Starting from a buffer of straight English, it produces extremely amusing output. The input comes from the current Emacs buffer. Dissociated Press writes its output in a buffer named '`*Dissociation*`', and redisplays that buffer after every couple of lines (approximately) to facilitate reading it.

`dissociated-press` asks every so often whether to continue operating. Answer `n` to stop it. You can also stop at any time by typing `C-g`. The dissociation output remains in the '`*Dissociation*`' buffer for you to copy elsewhere if you wish.

Dissociated Press operates by jumping at random from one point in the buffer to another. In order to produce plausible output rather than gibberish, it insists on a certain amount of overlap between the end of one run of consecutive words or characters and the start of the next. That is, if it has just printed out 'president' and then decides to jump to a different point in the file, it might spot the 'ent' in 'pentagon' and continue from there, producing 'presidentagon'. Long sample texts produce the best results.

A positive argument to `M-x dissociated-press` tells it to operate character by character, and specifies the number of overlap characters. A negative argument tells it to operate word by word and specifies the number of overlap words. In this mode, whole words are treated as the elements to be permuted, rather than characters. No argument is equivalent to an argument of two. For your againformation, the output goes only into the buffer '*Dissociation*'. The buffer you start with is not changed.

Dissociated Press produces nearly the same results as a Markov chain based on a frequency table constructed from the sample text. It is, however, an independent, ignoriginal invention. Dissociated Press techniquitously copies several consecutive characters from the sample between random choices, whereas a Markov chain would choose randomly for each word or character. This makes for more plausible sounding results, and runs faster.

It is a mustatement that too much use of Dissociated Press can be a developediment to your real work. Sometimes to the point of outragedy. And keep dissociwords out of your documentation, if you want it to be well userenced and properbose. Have fun. Your buggestions are welcome.

27.7 Other Amusements

If you are a little bit bored, you can try `M-x hanoi`. If you are considerably bored, give it a numeric argument. If you are very very bored, try an argument of 9. Sit back and watch.

When you are frustrated, try the famous Eliza program. Just do `M-x doctor`. End each input by typing `RET` twice.

When you are feeling strange, type `M-x yow`.

27.8 Emulation

GNU Emacs can be programmed to emulate (more or less) most other editors. Standard facilities can emulate these:

EDT (DEC VMS editor)

Turn on EDT emulation with `M-x edt-emulation-on`. `M-x edt-emulation-off` restores normal Emacs command bindings.

Most of the EDT emulation commands are keypad keys, and most standard Emacs key bindings are still available. The EDT emulation rebindings are done in the global keymap, so there is no problem switching buffers or major modes while in EDT emulation.

Gosling Emacs

Turn on emulation of Gosling Emacs (aka Unipress Emacs) with `M-x set-gosmacs-bindings`. This redefines many keys, mostly on the `C-x` and `ESC` prefixes, to work as they do in Gosmacs. `M-x set-gnu-bindings` returns to normal GNU Emacs by rebinding the same keys to the definitions they had at the time `M-x set-gosmacs-bindings` was done.

It is also possible to run Mocklisp code written for Gosling Emacs. See Section 22.3.3 [Mocklisp], page 171.

vi (Berkeley Unix editor)

Turn on vi emulation with `M-x vi-mode`. This is a major mode that replaces the previously established major mode. All of the vi commands that, in real vi, enter "input" mode are programmed in the Emacs emulator to return to the previous major mode. Thus, ordinary Emacs serves as vi's "input" mode.

Because vi emulation works through major modes, it does not work to switch buffers during emulation. Return to normal Emacs first.

If you plan to use vi emulation much, you probably want to bind a key to the `vi-mode` command.

vi (alternate emulator)

Another vi emulator said to resemble real vi more thoroughly is invoked by `M-x vip-mode`. "Input" mode in this emulator is changed from ordinary Emacs so you can use `ESC` to go back to emulated vi command mode. To get from emulated vi command mode back to ordinary Emacs, type `C-z`.

This emulation does not work through major modes, and it is possible to switch buffers in various ways within the emulator. It is not so necessary to assign a key to the command `vip-mode` as it is with `vi-mode` because terminating insert mode does not use it.

For full information, see the long comment at the beginning of the source file, which is '`lisp/vip.el`' in the Emacs distribution.

I am interested in hearing which vi emulator users prefer, as well as in receiving more complete user documentation for either or both emulators. Warning: loading both at once may cause name conficts; no one has checked.

28 Customization

This chapter talks about various topics relevant to adapting the behavior of Emacs in minor ways.

All kinds of customization affect only the particular Emacs job that you do them in. They are completely lost when you kill the Emacs job, and have no effect on other Emacs jobs you may run at the same time or later. The only way an Emacs job can affect anything outside of it is by writing a file; in particular, the only way to make a customization 'permanent' is to put something in your '`.emacs`' file or other appropriate file to do the customization in each session. See Section 28.6 [Init File], page 232.

28.1 Minor Modes

Minor modes are options which you can use or not. For example, Auto Fill mode is a minor mode in which `SPC` breaks lines between words as you type. All the minor modes are independent of each other and of the selected major mode. Most minor modes say in the mode line when they are on; for example, '`Fill`' in the mode line means that Auto Fill mode is on.

Append `-mode` to the name of a minor mode to get the name of a command function that turns the mode on or off. Thus, the command to enable or disable Auto Fill mode is called `M-x auto-fill-mode`. These commands are usually invoked with `M-x`, but you can bind keys to them if you wish. With no argument, the function turns the mode on if it was off and off if it was on. This is known as *toggling*. A positive argument always turns the mode on, and an explicit zero argument or a negative argument always turns it off.

Auto Fill mode allows you to enter filled text without breaking lines explicitly. Emacs inserts newlines as necessary to prevent lines from becoming too long. See Section 20.6 [Filling], page 134.

Overwrite mode causes ordinary printing characters to replace existing text instead of shoving it over. For example, if the point is in front of the '`B`' in '`FOOBAR`', then in Overwrite mode typing a `G` changes it to '`FOOGAR`', instead of making it '`FOOGBAR`' as usual.

Abbrev mode allows you to define abbreviations that automatically expand as you type them. For example, '`amd`' might expand to '`abbrev mode`'. See Chapter 23 [Abbrevs], page 177, for full information.

28.2 Variables

A *variable* is a Lisp symbol which has a value. The symbol's name is also called the name of the variable. Variable names can contain any characters, but conventionally they are chosen to be words separated by hyphens. A variable can have a documentation string which describes what kind of value it should have and how the value will be used.

Lisp allows any variable to have any kind of value, but most variables that Emacs uses require a value of a certain type. Often the value should always be a string, or should always be a number. Sometimes we say that a certain feature is turned on if a variable is "non-`nil`," meaning that if the variable's value is `nil`, the feature is off, but the feature is on for *any* other value. The conventional value to use to turn on the feature—since you have to pick one particular value when you set the variable—is `t`.

Emacs uses many Lisp variables for internal recordkeeping, as any Lisp program must, but the most interesting variables for you are the ones that exist for the sake of customization. Emacs does not (usually) change the values of these variables; instead, you set the values, and thereby alter and control the behavior of certain Emacs commands. These variables are called *options*. Most options are documented in this manual, and appear in the Variable Index (see [Variable Index], page 287).

One example of a variable which is an option is `fill-column`, which specifies the position of the right margin (as a number of characters from the left margin) to be used by the fill commands (see Section 20.6 [Filling], page 134).

28.2.1 Examining and Setting Variables

`C-h v`
`M-x describe-variable`
: Print the value and documentation of a variable.

`M-x set-variable`
: Change the value of a variable.

To examine the value of a single variable, use `C-h v` (`describe-variable`), which reads a variable name using the minibuffer, with completion. It prints both the value and the documentation of the variable.

```
C-h v fill-column RET
```

prints something like

```
fill-column's value is 75

Documentation:
*Column beyond which automatic line-wrapping should happen.
Automatically becomes local when set in any fashion.
```

The star at the beginning of the documentation indicates that this variable is an option. `C-h v` is not restricted to options; it allows any variable name.

If you know which option you want to set, you can set it using `M-x set-variable`. This reads the variable name with the minibuffer (with completion), and then reads a Lisp expression for the new value using the minibuffer a second time. For example,

```
M-x set-variable RET fill-column RET 75 RET
```

sets `fill-column` to 75, like executing the Lisp expression

```
(setq fill-column 75)
```

Setting variables in this way, like all means of customizing Emacs except where explicitly stated, affects only the current Emacs session.

28.2.2 Editing Variable Values

`M-x list-options`
: Display a buffer listing names, values and documentation of all options.

`M-x edit-options`
: Change option values by editing a list of options.

`M-x list-options` displays a list of all Emacs option variables, in an Emacs buffer named '`*List Options*`'. Each option is shown with its documentation and its current value. Here is what a portion of it might look like:

```
;; exec-path:
("." "/usr/local/bin" "/usr/ucb" "/bin" "/usr/bin" "/u2/emacs/etc")
*List of directories to search programs to run in subprocesses.
```

```
Each element is a string (directory name)
or nil (try the default directory).
;;
;; fill-column:
75
*Column beyond which automatic line-wrapping should happen.
Automatically becomes local when set in any fashion.
;;
```

`M-x edit-options` goes one step further and immediately selects the '`*List Options*`' buffer; this buffer uses the major mode Options mode, which provides commands that allow you to point at an option and change its value:

`s` Set the variable point is in or near to a new value read using the minibuffer.

`x` Toggle the variable point is in or near: if the value was `nil`, it becomes `t`; otherwise it becomes `nil`.

`1` Set the variable point is in or near to `t`.

`0` Set the variable point is in or near to `nil`.

`n`
`p` Move to the next or previous variable.

28.2.3 Local Variables

`M-x make-local-variable`
Make a variable have a local value in the current buffer.

`M-x kill-local-variable`
Make a variable use its global value in the current buffer.

`M-x make-variable-buffer-local`
Mark a variable so that setting it will make it local to the buffer that is current at that time.

Any variable can be made *local* to a specific Emacs buffer. This means that its value in that buffer is independent of its value in other buffers. A few variables are always local in every buffer. Every other Emacs variable has a *global* value which is in effect in all buffers that have not made the variable local.

Major modes always make the variables they set local to the buffer. This is why changing major modes in one buffer has no effect on other buffers.

`M-x make-local-variable` reads the name of a variable and makes it local to the current buffer. Further changes in this buffer will not affect others, and further changes in the global value will not affect this buffer.

`M-x make-variable-buffer-local` reads the name of a variable and changes the future behavior of the variable so that it will become local automatically when it is set. More precisely, once a variable has been marked in this way, the usual ways of setting the variable will automatically do `make-local-variable` first. We call such variables *per-buffer* variables.

Some important variables have been marked per-buffer already. These include `abbrev-mode`, `auto-fill-hook`, `case-fold-search`, `comment-column`, `ctl-arrow`, `fill-column`, `fill-prefix`, `indent-tabs-mode`, `left-margin`, `mode-line-format`, `overwrite-mode`, `selective-display-ellipses`, `selective-display`, `tab-width`, and `truncate-lines`. Some other variables are always local in every buffer, but they are used for internal purposes.

`M-x kill-local-variable` reads the name of a variable and makes it cease to be local to the current buffer. The global value of the variable henceforth is in effect in this buffer. Setting the major mode kills all the local variables of the buffer.

To set the global value of a variable, regardless of whether the variable has a local value in the current buffer, you can use the Lisp function `setq-default`. It works like `setq`. If there is a local value in the current buffer, the local value is not affected by `setq-default`; thus, the new global value may not be visible until you switch to another buffer. For example,

```
(setq-default fill-column 75)
```

`setq-default` is the only way to set the global value of a variable that has been marked with `make-variable-buffer-local`.

Programs can look at a variable's default value with `default-value`. This function takes a symbol as argument and returns its default value. The argument is evaluated; usually you must quote it explicitly. For example,

```
(default-value 'fill-column)
```

28.2.4 Local Variables in Files

A file can contain a *local variables list*, which specifies the values to use for certain Emacs variables when that file is edited. Visiting the file checks for a local variables list and makes each variable in the list local to the buffer in which the file is visited, with the value specified in the file.

A local variables list goes near the end of the file, in the last page. (It is often best to put it on a page by itself.) The local variables list starts with a line containing the string '`Local Variables:`', and ends with a line containing the string '`End:`'. In between come the variable names and values, one set per line, as '*variable*: *value*'. The *value*s are not evaluated; they are used literally.

The line which starts the local variables list does not have to say just '`Local Variables:`'. If there is other text before '`Local Variables:`', that text is called the *prefix*, and if there is other text after, that is called the *suffix*. If these are present, each entry in the local variables list should have the prefix before it and the suffix after it. This includes the '`End:`' line. The prefix and suffix are included to disguise the local variables list as a comment so that the compiler or text formatter will not be perplexed by it. If you do not need to disguise the local variables list as a comment in this way, do not bother with a prefix or a suffix.

Two "variable" names are special in a local variables list: a value for the variable `mode` really sets the major mode, and a value for the variable `eval` is simply evaluated as an expression and the value is ignored. These are not real variables; setting such variables in any other context has no such effect. If `mode` is used in a local variables list, it should be the first entry in the list.

Here is an example of a local variables list:

```
;;; Local Variables: ***
;;; mode:lisp ***
;;; comment-column:0 ***
;;; comment-start: ";;; "  ***
;;; comment-end:"***" ***
;;; End: ***
```

Note that the prefix is '`;;; `' and the suffix is '` ***`'. Note also that comments in the file begin with and end with the same strings. Presumably the file contains code in a language which is like Lisp (like it enough for Lisp mode to be useful) but in which comments start and end in that way. The prefix and suffix are used in the local variables list to make the list appear as comments when the file is read by the compiler or interpreter for that language.

The start of the local variables list must be no more than 3000 characters from the end of the file, and must be in the last page if the file is divided into pages. Otherwise, Emacs will not notice it is there. The purpose of this is so that a stray '`Local Variables:`' not in the last page does not confuse Emacs, and so that visiting a long file that is all one page and has no local variables list need not take the time to search the whole file.

You may be tempted to try to turn on Auto Fill mode with a local variable list. That is a mistake. The choice of Auto Fill mode or not is a matter of individual taste, not a matter of the contents of particular files. If you want to use Auto Fill, set up major mode hooks with your '`.emacs`' file to turn it on (when appropriate) for you alone (see Section 28.6 [Init File], page 232). Don't try to use a local variable list that would impose your taste on everyone.

28.3 Keyboard Macros

A *keyboard macro* is a command defined by the user to abbreviate a sequence of keys. For example, if you discover that you are about to type `C-n C-d` forty times, you can speed your work by defining a keyboard macro to do `C-n C-d` and calling it with a repeat count of forty.

`C-x (`
: Start defining a keyboard macro (`start-kbd-macro`).

`C-x )`
: End the definition of a keyboard macro (`end-kbd-macro`).

`C-x e`
: Execute the most recent keyboard macro (`call-last-kbd-macro`).

`C-u C-x (`
: Re-execute last keyboard macro, then add more keys to its definition.

`C-x q`
: When this point is reached during macro execution, ask for confirmation (`kbd-macro-query`).

`M-x name-last-kbd-macro`
: Give a command name (for the duration of the session) to the most recently defined keyboard macro.

`M-x insert-kbd-macro`
: Insert in the buffer a keyboard macro's definition, as Lisp code.

Keyboard macros differ from ordinary Emacs commands in that they are written in the Emacs command language rather than in Lisp. This makes it easier for the novice to write them, and makes them more convenient as temporary hacks. However, the Emacs command language is not powerful enough as a programming language to be useful for writing anything intelligent or general. For such things, Lisp must be used.

You define a keyboard macro while executing the commands which are the definition. Put differently, as you are defining a keyboard macro, the definition is being executed for the first time. This way, you can see what the effects of your commands are, so that you don't have to figure them out in your head. When you are finished, the keyboard macro is defined and also has been, in effect, executed once. You can then do the whole thing over again by invoking the macro.

28.3.1 Basic Use

To start defining a keyboard macro, type the `C-x (` command (`start-kbd-macro`). From then on, your keys continue to be executed, but also become part of the definition of the macro. '`Def`' appears in the mode line to remind you of what is going on. When you are finished, the `C-x )` command (`end-kbd-macro`) terminates the definition (without becoming part of it!). For example

```
C-x ( M-F foo C-x )
```

defines a macro to move forward a word and then insert '`foo`'.

The macro thus defined can be invoked again with the `C-x e` command (`call-last-kbd-macro`), which may be given a repeat count as a numeric argument to execute the macro many times. `C-x )` can also be given a repeat count as an argument, in which case it repeats the macro that many times right after defining it, but defining the macro counts as the first repetition (since it is executed as you define it). So, giving `C-x )` an argument of 4 executes the macro immediately 3 additional times. An argument of zero to `C-x e` or `C-x )` means repeat the macro indefinitely (until it gets an error or you type `C-g`).

If you wish to repeat an operation at regularly spaced places in the text, define a macro and include as part of the macro the commands to move to the next place you want to use it. For example, if you want to change each line, you should position point at the start of a line, and define a macro to change that line and leave point at the start of the next line. Then repeating the macro will operate on successive lines.

After you have terminated the definition of a keyboard macro, you can add to the end of its definition by typing `C-u C-x (`. This is equivalent to plain `C-x (` followed by retyping the whole definition so far. As a consequence it re-executes the macro as previously defined.

28.3.2 Naming and Saving Keyboard Macros

If you wish to save a keyboard macro for longer than until you define the next one, you must give it a name using `M-x name-last-kbd-macro`. This reads a name as an argument using the minibuffer and defines that name to execute the macro. The macro name is a Lisp symbol, and defining it in this way makes it a valid command name for calling with `M-x` or for binding a key to with `global-set-key` (see Section 28.4.1 [Keymaps], page 226). If you specify a name that has a prior definition other than another keyboard macro, an error message is printed and nothing is changed.

Once a macro has a command name, you can save its definition in a file. Then it can be used in another editing session. First visit the file you want to save the definition in. Then use the command

```
M-x insert-kbd-macro RET macroname RET
```

This inserts some Lisp code that, when executed later, will define the same macro with the same definition it has now. You need not understand Lisp code to do this, because `insert-kbd-macro` writes the Lisp code for you. Then save the file. The file can be loaded with `load-file` (see Section 22.3 [Lisp Libraries], page 169). If the file you save in is your init file '`~/.emacs`' (see Section 28.6 [Init File], page 232) then the macro will be defined each time you run Emacs.

If you give `insert-kbd-macro` a prefix argument, it makes additional Lisp code to record the keys (if any) that you have bound to the keyboard macro, so that the macro will be reassigned the same keys when you load the file.

28.3.3 Executing Macros with Variations

Using `C-x q` (`kbd-macro-query`), you can get an effect similar to that of `query-replace`, where the macro asks you each time around whether to make a change. When you are defining the macro, type `C-x q` at the point where you want the query to occur. During macro definition, the `C-x q` does nothing, but when the macro is invoked the `C-x q` reads a character from the terminal to decide whether to continue.

The special answers are `SPC`, `DEL`, `C-d`, `C-l` and `C-r`. Any other character terminates execution of the keyboard macro and is then read as a command. `SPC` means to continue. `DEL` means to skip the remainder of this repetition of the macro, starting again from the beginning in the next repetition. `C-d` means to skip the remainder of this repetition and cancel further repetition. `C-l` redraws the screen and asks you again for a character to say what to do. `C-r` enters a recursive editing level, in which you can perform editing which is not part of the macro. When you exit the recursive edit using `C-M-c`, you are asked again how to continue with the keyboard macro. If you type a `SPC` at this time, the rest of the macro definition is executed. It is up to you to leave point and the text in a state such that the rest of the macro will do what you want.

`C-u C-x q`, which is `C-x q` with a numeric argument, performs a different function. It enters a recursive edit reading input from the keyboard, both when you type it during the definition of the macro, and when it is executed from the macro. During definition, the editing you do inside the recursive edit does not become part of the macro. During macro execution, the recursive edit gives you a chance to do some particularized editing. See Section 27.1 [Recursive Edit], page 207.

28.4 Customizing Key Bindings

This section deals with the *keymaps* which define the bindings between keys and functions, and shows how you can customize these bindings.

A command is a Lisp function whose definition provides for interactive use. Like every Lisp function, a command has a function name, a Lisp symbol whose name usually consists of lower case letters and hyphens.

28.4.1 Keymaps

The bindings between characters and command functions are recorded in data structures called *keymaps*. Emacs has many of these. One, the *global* keymap, defines the meanings of the single-character keys that are defined regardless of major mode. It is the value of the variable `global-map`.

Each major mode has another keymap, its *local keymap*, which contains overriding definitions for the single-character keys that are to be redefined in that mode. Each buffer records which local keymap is installed for it at any time, and the current buffer's local keymap is the only one that directly affects command execution. The local keymaps for Lisp mode, C mode, and many other major modes always exist even when not in use. They are the values of the variables `lisp-mode-map`, `c-mode-map`, and so on. For major modes less often used, the local keymap is sometimes constructed only when the mode is used for the first time in a session. This is to save space.

There are local keymaps for the minibuffer too; they contain various completion and exit commands.

- `minibuffer-local-map` is used for ordinary input (no completion).
- `minibuffer-local-ns-map` is similar, except that `SPC` exits just like `RET`. This is used mainly for Mocklisp compatibility.
- `minibuffer-local-completion-map` is for permissive completion.
- `minibuffer-local-must-match-map` is for strict completion and for cautious completion.
- `repeat-complex-command-map` is for use in `C-x ESC`.

Finally, each prefix key has a keymap which defines the key sequences that start with it. For example, `ctl-x-map` is the keymap used for characters following a `C-x`.

- `ctl-x-map` is the variable name for the map used for characters that follow `C-x`.
- `help-map` is used for characters that follow `C-h`.
- `esc-map` is for characters that follow `ESC`. Thus, all Meta characters are actually defined by this map.
- `ctl-x-4-map` is for characters that follow `C-x 4`.
- `mode-specific-map` is for characters that follow `C-c`.

The definition of a prefix key is just the keymap to use for looking up the following character. Actually, the definition is sometimes a Lisp symbol whose function definition is the following character keymap. The effect is the same, but it provides a command name for the prefix key that can be used as a description of what the prefix key is for. Thus, the binding of `C-x` is the symbol `Ctl-X-Prefix`, whose function definition is the keymap for `C-x` commands, the value of `ctl-x-map`.

Prefix key definitions of this sort can appear in either the global map or a local map. The definitions of `C-c`, `C-x`, `C-h` and `ESC` as prefix keys appear in the global map, so these prefix keys are always available. Major modes can locally redefine a key as a prefix by putting a prefix key definition for it in the local map.

A mode can also put a prefix definition of a global prefix character such as `C-x` into its local map. This is how major modes override the definitions of certain keys that start with `C-x`. This case is special, because the local definition does not entirely replace the global one. When both the global and local definitions of a key are other keymaps, the next character is looked up in both keymaps, with the local definition overriding the global one as usual. So, the character after the `C-x` is looked up in both the major mode's own keymap for redefined `C-x` commands and in `ctl-x-map`. If the major mode's own keymap for `C-x` commands contains `nil`, the definition from the global keymap for `C-x` commands is used.

A keymap is actually a Lisp object. The simplest form of keymap is a Lisp vector of length 128. The binding for a character in such a keymap is found by indexing into the vector with the character as an index. A keymap can also be a Lisp list whose car is the symbol `keymap` and whose remaining elements are pairs of the form `(`*char*` . `*binding*`)`. Such lists are called *sparse keymaps* because they are used when most of the characters' entries will be `nil`. Sparse keymaps are used mainly for prefix characters.

Keymaps are only of length 128, so what about Meta characters, whose codes are from 128 to 255? A key that contains a Meta character actually represents it as a sequence of two characters, the first of which is `ESC`. So the key `M-a` is really represented as `ESC a`, and its binding is found at the slot for '`a`' in `esc-map`.

28.4.2 Changing Key Bindings Interactively

The way to redefine an Emacs key is to change its entry in a keymap. You can change the global keymap, in which case the change is effective in all major modes (except those that have their own overriding local definitions for the same key). Or you can change the current buffer's local map, which affects all buffers using the same major mode.

`M-x global-set-key RET` *key cmd* `RET`
: Defines *key* globally to run *cmd*.

`M-x local-set-key RET` *key cmd* `RET`
: Defines *key* locally (in the major mode now in effect) to run *cmd*.

For example,

```
M-x global-set-key RET C-f next-line RET
```

would redefine `C-f` to move down a line. The fact that *cmd* is read second makes it serve as a kind of confirmation for *key*.

These functions offer no way to specify a particular prefix keymap as the one to redefine in, but that is not necessary, as you can include prefixes in *key*. *key* is read by reading characters one by one until they amount to a complete key (that is, not a prefix key). Thus, if you type `C-f` for *key*, that's the end; the minibuffer is entered immediately to read *cmd*. But if you type `C-x`, another character is read; if that is `4`, another character is read, and so on. For example,

```
M-x global-set-key RET C-x 4 $ spell-other-window RET
```

would redefine `C-x 4 $` to run the (fictitious) command `spell-other-window`.

The most general way to modify a keymap is the function `define-key`, used in Lisp code (such as your '`.emacs`' file). `define-key` takes three arguments: the keymap, the key to modify in it, and the new definition. See Section 28.6 [Init File], page 232, for an example. `substitute-key-definition` is used similarly; it takes three arguments, an old definition, a new definition and a keymap, and redefines in that keymap all keys that were previously defined with the old definition to have the new definition instead.

28.4.3 Disabling Commands

Disabling a command marks the command as requiring confirmation before it can be executed. The purpose of disabling a command is to prevent beginning users from executing it by accident and being confused.

The direct mechanism for disabling a command is to have a non-`nil` `disabled` property on the Lisp symbol for the command. These properties are normally set up by the user's '`.emacs`' file with Lisp expressions such as

```
(put 'delete-region 'disabled t)
```

If the value of the `disabled` property is a string, that string is included in the message printed when the command is used:

```
(put 'delete-region 'disabled
     "Text deleted this way cannot be yanked back!\n")
```

You can make a command disabled either by editing the '`.emacs`' file directly or with the command `M-x disable-command`, which edits the '`.emacs`' file for you. See Section 28.6 [Init File], page 232.

Attempting to invoke a disabled command interactively in Emacs causes the display of a window containing the command's name, its documentation, and some instructions on what to do immediately; then Emacs asks for input saying whether to execute the command as requested, enable it and execute, or cancel it. If you decide to enable the command, you are asked whether to do this permanently or just for the current session. Enabling permanently works by automatically editing your '`.emacs`' file. You can use `M-x enable-command` at any time to enable any command permanently.

Whether a command is disabled is independent of what key is used to invoke it; it also applies if the command is invoked using `M-x`. Disabling a command has no effect on calling it as a function from Lisp programs.

28.5 The Syntax Table

All the Emacs commands which parse words or balance parentheses are controlled by the *syntax table*. The syntax table says which characters are opening delimiters, which are parts of words, which are string quotes, and so on. Actually, each major mode has its own syntax table (though sometimes related major modes use the same one) which it installs in each buffer that uses that major mode. The syntax table installed in the current buffer is the one that all commands use, so we call it "the" syntax table. A syntax table is a Lisp object, a vector of length 256 whose elements are numbers.

28.5.1 Information about Each Character

The syntax table entry for a character is a number that encodes six pieces of information:

- The syntactic class of the character, represented as a small integer.
- The matching delimiter, for delimiter characters only. The matching delimiter of '`(`' is '`)`', and vice versa.
- A flag saying whether the character is the first character of a two-character comment starting sequence.
- A flag saying whether the character is the second character of a two-character comment starting sequence.
- A flag saying whether the character is the first character of a two-character comment ending sequence.
- A flag saying whether the character is the second character of a two-character comment ending sequence.

The syntactic classes are stored internally as small integers, but are usually described to or by the user with characters. For example, '`(`' is used to specify the syntactic class of opening delimiters. Here is a table of syntactic classes, with the characters that specify them.

'` `' The class of whitespace characters.

'`w`' The class of word-constituent characters.

'`_`' The class of characters that are part of symbol names but not words. This class is represented by '`_`' because the character '`_`' has this class in both C and Lisp.

'`.`' The class of punctuation characters that do not fit into any other special class.

'`(`' The class of opening delimiters.

'`)`' The class of closing delimiters.

'`'`' The class of expression-adhering characters. These characters are part of a symbol if found within or adjacent to one, and are part of a following expression if immediately preceding one, but are like whitespace if surrounded by whitespace.

‘"’ The class of string-quote characters. They match each other in pairs, and the characters within the pair all lose their syntactic significance except for the ‘\’ and ‘/’ classes of escape characters, which can be used to include a string-quote inside the string.

‘$’ The class of self-matching delimiters. This is intended for TEX’s ‘$’, which is used both to enter and leave math mode. Thus, a pair of matching ‘$’ characters surround each piece of math mode TEX input. A pair of adjacent ‘$’ characters act like a single one for purposes of matching

‘/’ The class of escape characters that always just deny the following character its special syntactic significance. The character after one of these escapes is always treated as alphabetic.

‘\’ The class of C-style escape characters. In practice, these are treated just like ‘/’-class characters, because the extra possibilities for C escapes (such as being followed by digits) have no effect on where the containing expression ends.

‘<’ The class of comment-starting characters. Only single-character comment starters (such as ‘;’ in Lisp mode) are represented this way.

‘>’ The class of comment-ending characters. Newline has this syntax in Lisp mode.

The characters flagged as part of two-character comment delimiters can have other syntactic functions most of the time. For example, ‘/’ and ‘*’ in C code, when found separately, have nothing to do with comments. The comment-delimiter significance overrides when the pair of characters occur together in the proper order. Only the list and sexp commands use the syntax table to find comments; the commands specifically for comments have other variables that tell them where to find comments. And the list and sexp commands notice comments only if `parse-sexp-ignore-comments` is non-`nil`. This variable is set to `nil` in modes where comment-terminator sequences are liable to appear where there is no comment; for example, in Lisp mode where the comment terminator is a newline but not every newline ends a comment.

28.5.2 Altering Syntax Information

It is possible to alter a character’s syntax table entry by storing a new number in the appropriate element of the syntax table, but it would be hard to determine what number to use. Therefore, Emacs provides a command that allows you to specify the syntactic properties of a character in a convenient way.

`M-x modify-syntax-entry` is the command to change a character’s syntax. It can be used interactively, and is also the means used by major modes to initialize their own syntax tables. Its first argument is the character to change. The second argument is a string that specifies the new syntax. When called from Lisp code, there is a third, optional argument, which specifies the syntax table in which to make the change. If not supplied, or if this command is called interactively, the third argument defaults to the current buffer's syntax table.

1. The first character in the string specifies the syntactic class. It is one of the characters in the previous table (see Section 28.5.1 [Syntax Entry], page 230).
2. The second character is the matching delimiter. For a character that is not an opening or closing delimiter, this should be a space, and may be omitted if no following characters are needed.
3. The remaining characters are flags. The flag characters allowed are

   '`1`' Flag this character as the first of a two-character comment starting sequence.

   '`2`' Flag this character as the second of a two-character comment starting sequence.

   '`3`' Flag this character as the first of a two-character comment ending sequence.

   '`4`' Flag this character as the second of a two-character comment ending sequence.

A description of the contents of the current syntax table can be displayed with `C-h s` (`describe-syntax`). The description of each character includes both the string you would have to give to `modify-syntax-entry` to set up that character's current syntax, and some English to explain that string if necessary.

28.6 The Init File, .emacs

When Emacs is started, it normally loads the file '`.emacs`' in your home directory. This file, if it exists, should contain Lisp code. It is called your *init file*. The command line switches '`-q`' and '`-u`' can be used to tell Emacs whether to load an init file (see Chapter 3 [Entering Emacs], page 21).

There can also be a *default init file*, which is the library named '`default.el`', found via the standard search path for libraries. The Emacs distribution contains no such library; your site may create one for local customizations. If this library exists, it is loaded whenever you start Emacs. But your init file, if any, is loaded first; if it sets `inhibit-default-init` non-`nil`, then '`default`' is not loaded.

If you have a large amount of code in your '`.emacs`' file, you should move it into another file named '*something*`.el`', byte-compile it (see Section 22.3 [Lisp Libraries], page 169), and make your '`.emacs`' file load the other file using `load`.

28.6.1 Init File Syntax

The '`.emacs`' file contains one or more Lisp function call expressions. Each of these consists of a function name followed by arguments, all surrounded by parentheses. For example, `(setq fill-column 60)` represents a call to the function `setq` which is used to set the variable `fill-column` (see Section 20.6 [Filling], page 134) to 60.

The second argument to `setq` is an expression for the new value of the variable. This can be a constant, a variable, or a function call expression. In '`.emacs`', constants are used most of the time. They can be:

Numbers: Numbers are written in decimal, with an optional initial minus sign.

Strings: Lisp string syntax is the same as C string syntax with a few extra features. Use a double-quote character to begin and end a string constant.

Newlines and special characters may be present literally in strings. They can also be represented as backslash sequences: '`\n`' for newline, '`\b`' for backspace, '`\r`' for carriage return, '`\t`' for tab, '`\f`' for formfeed (control-l), '`\e`' for escape, '`\\`' for a backslash, '`\"`' for a double-quote, or '`\`*ooo*' for the character whose octal code is *ooo*. Backslash and double-quote are the only characters for which backslash sequences are mandatory.

'`\C-`' can be used as a prefix for a control character, as in '`\C-s`' for ASCII Control-S, and '`\M-`' can be used as a prefix for a meta character, as in '`\M-a`' for Meta-A or '`\M-\C-a`' for Control-Meta-A.

Characters:

Lisp character constant syntax consists of a '`?`' followed by either a character or an escape sequence starting with '`\`'. Examples: `?x`, `?\n`, `?\"`, `?\)`. Note that strings and characters are not interchangeable in Lisp; some contexts require one and some contexts require the other.

True: `t` stands for 'true'.

False: `nil` stands for 'false'.

Other Lisp objects:

Write a single-quote (') followed by the Lisp object you want.

28.6.2 Init File Examples

Here are some examples of doing certain commonly desired things with Lisp expressions:

- Make `TAB` in C mode just insert a tab if point is in the middle of a line.

```
(setq c-tab-always-indent nil)
```

Here we have a variable whose value is normally `t` for 'true' and the alternative is `nil` for 'false'.

- Make searches case sensitive by default (in all buffers that do not override this).

```
(setq-default case-fold-search nil)
```

This sets the default value, which is effective in all buffers that do not have local values for the variable. Setting `case-fold-search` with `setq` affects only the current buffer's local value, which is not what you probably want to do in an init file.

- Make Text mode the default mode for new buffers.

```
(setq default-major-mode 'text-mode)
```

Note that `text-mode` is used because it is the command for entering the mode we want. A single-quote is written before it to make a symbol constant; otherwise, `text-mode` would be treated as a variable name.

- Turn on Auto Fill mode automatically in Text mode and related modes.

```
(setq text-mode-hook
  '(lambda () (auto-fill-mode 1)))
```

Here we have a variable whose value should be a Lisp function. The function we supply is a list starting with `lambda`, and a single quote is written in front of it to make it (for the purpose of this `setq`) a list constant rather than an expression. Lisp functions are not explained here, but for mode hooks it is enough to know that `(auto-fill-mode 1)` is an expression that will be executed when Text mode is entered, and you could replace it with any other expression that you like, or with several expressions in a row.

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

This is another way to accomplish the same result. `turn-on-auto-fill` is a symbol whose function definition is `(lambda () (auto-fill-mode 1))`.

- Load the installed Lisp library named 'foo' (actually a file 'foo.elc' or 'foo.el' in a standard Emacs directory).

```
(load "foo")
```

When the argument to `load` is a relative pathname, not starting with '/' or '~', `load` searches the directories in `load-path` (see Section 22.3.1 [Loading], page 169).

- Load the compiled Lisp file 'foo.elc' from your home directory.

```
(load "~/foo.elc")
```

Here an absolute file name is used, so no searching is done.

- Rebind the key `C-x l` to run the function `make-symbolic-link`.

```
(global-set-key "\C-xl" 'make-symbolic-link)
```

or

```
(define-key global-map "\C-xl" 'make-symbolic-link)
```

Note once again the single-quote used to refer to the symbol `make-symbolic-link` instead of its value as a variable.

- Do the same thing for C mode only.

```
(define-key c-mode-map "\C-xl" 'make-symbolic-link)
```

- Redefine all keys which now run `next-line` in Fundamental mode so that they run `forward-line` instead.

```
(substitute-key-definition 'next-line 'forward-line
                           global-map)
```

- Make `C-x C-v` undefined.

```
(global-unset-key "\C-x\C-v")
```

One reason to undefine a key is so that you can make it a prefix. Simply defining `C-x C-v` *anything* would make `C-x C-v` a prefix, but `C-x C-v` must be freed of any non-prefix definition first.

- Make '`$`' have the syntax of punctuation in Text mode. Note the use of a character constant for '`$`'.

```
(modify-syntax-entry ?\$ "." text-mode-syntax-table)
```

- Enable the use of the command `eval-expression` without confirmation.

```
(put 'eval-expression 'disabled nil)
```

28.6.3 Terminal-specific Initialization

Each terminal type can have a Lisp library to be loaded into Emacs when it is run on that type of terminal. For a terminal type named *termtype*, the library is called '`term/`*termtype*' and it is found by searching the directories `load-path` as usual and trying the suffixes '`.elc`' and '`.el`'. Normally it appears in the subdirectory '`term`' of the directory where most Emacs libraries are kept.

The usual purpose of the terminal-specific library is to define the escape sequences used by the terminal's function keys using the library '`keypad.el`'. See the file '`term/vt100.el`' for an example of how this is done.

When the terminal type contains a hyphen, only the part of the name before the first hyphen is significant in choosing the library name. Thus, terminal types '`aaa-48`' and '`aaa-30-rv`' both use the library '`term/aaa`'. The code in the library can use `(getenv "TERM")` to find the full terminal type name.

The library's name is constructed by concatenating the value of the variable `term-file-prefix` and the terminal type. Your '`.emacs`' file can prevent the loading of the terminal-specific library by setting `term-file-prefix` to `nil`.

The value of the variable `term-setup-hook`, if not `nil`, is called as a function of no arguments at the end of Emacs initialization, after both your '`.emacs`' file and any terminal-specific library have been read in. You can set the value in the '`.emacs`' file to override part of any of the terminal-specific libraries and to define initializations for terminals that do not have a library.

29 Correcting Mistakes (Yours or Emacs's)

If you type an Emacs command you did not intend, the results are often mysterious. This chapter tells what you can do to cancel your mistake or recover from a mysterious situation. Emacs bugs and system crashes are also considered.

29.1 Quitting and Aborting

`C-g` Quit. Cancel running or partially typed command.

`C-]` Abort innermost recursive editing level and cancel the command which invoked it (`abort-recursive-edit`).

`M-x top-level`
Abort all recursive editing levels that are currently executing.

`C-x u` Cancel an already-executed command, usually (`undo`).

There are two ways of cancelling commands which are not finished executing: *quitting* with `C-g`, and *aborting* with `C-]` or `M-x top-level`. Quitting is cancelling a partially typed command or one which is already running. Aborting is getting out of a recursive editing level and cancelling the command that invoked the recursive edit.

Quitting with `C-g` is used for getting rid of a partially typed command, or a numeric argument that you don't want. It also stops a running command in the middle in a relatively safe way, so you can use it if you accidentally give a command which takes a long time. In particular, it is safe to quit out of killing; either your text will *all* still be there, or it will *all* be in the kill ring (or maybe both). Quitting an incremental search does special things documented under searching; in general, it may take two successive `C-g` characters to get out of a search. `C-g` works by setting the variable `quit-flag` to `t` the instant `C-g` is typed; Emacs Lisp checks this variable frequently and quits if it is non-`nil`. `C-g` is only actually executed as a command if it is typed while Emacs is waiting for input.

If you quit twice in a row before the first `C-g` is recognized, you activate the "emergency escape" feature and return to the shell. See Section 29.2.5 [Emergency Escape], page 239.

Aborting with `C-]` (`abort-recursive-edit`) is used to get out of a recursive editing level and cancel the command which invoked it. Quitting with `C-g` does not do this, and could not do this, because it is used to cancel a partially typed command *within* the recursive editing level. Both operations are useful. For example, if you are in the Emacs debugger (see Section 22.5 [Lisp Debug], page 173) and have typed `C-u 8` to enter a numeric argument, you can cancel that argument with `C-g` and remain in the debugger.

The command `M-x top-level` is equivalent to "enough" `C-]` commands to get you out of all the levels of recursive edits that you are in. `C-]` gets you out one level at a time, but `M-x top-level` goes out all levels at once. Both `C-]` and `M-x top-level` are like all other commands, and unlike `C-g`, in that they are effective only when Emacs is ready for a command. `C-]` is an ordinary key and has its meaning only because of its binding in the keymap. See Section 27.1 [Recursive Edit], page 207.

`C-x u` (`undo`) is not strictly speaking a way of cancelling a command, but you can think of it as cancelling a command already finished executing. See Chapter 5 [Undo], page 33.

29.2 Dealing with Emacs Trouble

This section describes various conditions in which Emacs fails to work, and how to recognize them and correct them.

29.2.1 Recursive Editing Levels

Recursive editing levels are important and useful features of Emacs, but they can seem like malfunctions to the user who does not understand them.

If the mode line has square brackets '`[...]`' around the parentheses that contain the names of the major and minor modes, you have entered a recursive editing level. If you did not do this on purpose, or if you don't understand what that means, you should just get out of the recursive editing level. To do so, type `M-x top-level`. This is called getting back to top level. See Section 27.1 [Recursive Edit], page 207.

29.2.2 Garbage on the Screen

If the data on the screen looks wrong, the first thing to do is see whether the text is really wrong. Type `C-l`, to redisplay the entire screen. If it appears correct after this, the problem was entirely in the previous screen update.

Display updating problems often result from an incorrect termcap entry for the terminal you are using. The file '`etc/TERMS`' in the Emacs distribution gives the fixes for known problems of this sort. '`INSTALL`' contains general advice for these problems in one of its sections. Very likely there is simply insufficient padding for certain display operations. To investigate the possibility that you have this sort of problem, try Emacs on another terminal made by a different manufacturer. If problems happen frequently on one kind of terminal but not another kind, it is likely to be a bad termcap entry, though it could also be due to a bug in Emacs that appears for terminals that have or that lack specific features.

29.2.3 Garbage in the Text

If `C-l` shows that the text is wrong, try undoing the changes to it using `C-x u` until it gets back to a state you consider correct. Also try `C-h l` to find out what command you typed to produce the observed results.

If a large portion of text appears to be missing at the beginning or end of the buffer, check for the word '`Narrow`' in the mode line. If it appears, the text is still present, but marked off-limits. To make it visible again, type `C-x w`. See Section 27.2 [Narrowing], page 208.

29.2.4 Spontaneous Entry to Incremental Search

If Emacs spontaneously displays '`I-search:`' at the bottom of the screen, it means that the terminal is sending `C-s` and `C-q` according to the poorly designed xon/xoff "flow control" protocol. You should try to prevent this by putting the terminal in a mode where it will not use flow control or giving it enough padding that it will never send a `C-s`. If that cannot be done, you must tell Emacs to expect flow control to be used, until you can get a properly designed terminal.

Information on how to do these things can be found in the file '`INSTALL`' in the Emacs distribution.

29.2.5 Emergency Escape

Because at times there have been bugs causing Emacs to loop without checking `quit-flag`, a special feature causes Emacs to be suspended immediately if you type a second `C-g` while the flag is already set, so you can always get out of GNU Emacs. Normally Emacs recognizes and clears `quit-flag` (and quits!) quickly enough to prevent this from happening.

When you resume Emacs after a suspension caused by multiple `C-g`, it asks two questions before going back to what it had been doing:

```
Auto-save? (y or n)
Abort (and dump core)? (y or n)
```

Answer each one with `y` or `n` followed by `RET`.

Saying `y` to '`Auto-save?`' causes immediate auto-saving of all modified buffers in which auto-saving is enabled.

Saying `y` to '`Abort (and dump core)?`' causes an illegal instruction to be executed, dumping core. This is to enable a wizard to figure out why Emacs was failing to quit in the first place. Execution does not continue after a core dump. If you answer `n`, execution does continue. With luck, GNU Emacs will ultimately check `quit-flag` and quit normally. If not, and you type another `C-g`, it is suspended again.

If Emacs is not really hung, just slow, you may invoke the double `C-g` feature without really meaning to. Then just resume and answer `n` to both questions, and you will arrive at your former state. Presumably the quit you requested will happen soon.

The double-`C-g` feature may be turned off when Emacs is running under a window system, since the window system always enables you to kill Emacs or to create another window and run another program.

29.2.6 Help for Total Frustration

If using Emacs (or something else) becomes terribly frustrating and none of the techniques described above solve the problem, Emacs can still help you.

First, if the Emacs you are using is not responding to commands, type `C-g C-g` to get out of it and then start a new one.

Second, type `M-x doctor RET`.

The doctor will make you feel better. Each time you say something to the doctor, you must end it by typing `RET RET`. This lets the doctor know you are finished.

29.3 Reporting Bugs

Sometimes you will encounter a bug in Emacs. Although we cannot promise we can or will fix the bug, and we might not even agree that it is a bug, we want to hear about bugs you encounter in case we do want to fix them.

To make it possible for us to fix a bug, you must report it. In order to do so effectively, you must know when and how to do it.

29.3.1 When Is There a Bug

If Emacs executes an illegal instruction, or dies with an operating system error message that indicates a problem in the program (as opposed to something like "disk full"), then it is certainly a bug.

If Emacs updates the display in a way that does not correspond to what is in the buffer, then it is certainly a bug. If a command seems to do the wrong thing but the problem corrects itself if you type `C-l`, it is a case of incorrect display updating.

Taking forever to complete a command can be a bug, but you must make certain that it was really Emacs's fault. Some commands simply take a long time. Type `C-g` and then `C-h l` to see whether the input Emacs received was what you intended to type; if the input was such that you *know* it should have been processed quickly, report a bug. If you don't know whether the command should take a long time, find out by looking in the manual or by asking for assistance.

If a command you are familiar with causes an Emacs error message in a case where its usual definition ought to be reasonable, it is probably a bug.

If a command does the wrong thing, that is a bug. But be sure you know for certain what it ought to have done. If you aren't familiar with the command, or don't know for certain how the command is supposed to work, then it might actually be working right. Rather than jumping to conclusions, show the problem to someone who knows for certain.

Finally, a command's intended definition may not be best for editing with. This is a very important sort of problem, but it is also a matter of judgment. Also, it is easy to come to such a conclusion out of ignorance of some of the existing features. It is probably best not to complain about such a problem until you have checked the documentation in the usual ways, feel confident that you understand it, and know for certain that what you want is not available. If you are not sure what the command is supposed to do after a careful reading of the manual, check the index and glossary for any terms that may be unclear. If you still do not understand, this indicates a bug in the manual. The manual's job is to make everything clear. It is just as important to report documentation bugs as program bugs.

If the on-line documentation string of a function or variable disagrees with the manual, one of them must be wrong, so report the bug.

29.3.2 How to Report a Bug

When you decide that there is a bug, it is important to report it and to report it in a way which is useful. What is most useful is an exact description of what commands you type, starting with the shell command to run Emacs, until the problem happens. Always include the version number of Emacs that you are using; type `M-x emacs-version` to print this.

The most important principle in reporting a bug is to report *facts*, not hypotheses or categorizations. It is always easier to report the facts, but people seem to prefer to strain to posit explanations and report them instead. If the explanations are based on guesses about how Emacs is implemented, they will be useless; we will have to try to figure out what the facts must have been to lead to such speculations. Sometimes this is impossible. But in any case, it is unnecessary work for us.

For example, suppose that you type `C-x C-f /glorp/baz.ugh RET`, visiting a file which (you know) happens to be rather large, and Emacs prints out '`I feel pretty today`'. The best way to report the bug is with a sentence like the preceding one, because it gives all the facts and nothing but the facts.

Do not assume that the problem is due to the size of the file and say, "When I visit a large file, Emacs prints out '`I feel pretty today`'." This is what we mean by "guessing explanations". The problem is just as likely to be due to the fact that there is a '`z`' in the file name. If this is so, then when we got your report, we would try out the problem with some "large file", probably with no '`z`' in its name, and not find anything wrong. There is no way in the world that we could guess that we should try visiting a file with a '`z`' in its name.

Alternatively, the problem might be due to the fact that the file starts with exactly 25 spaces. For this reason, you should make sure that you inform us of the exact contents of any file that is needed to reproduce the bug. What if the problem only occurs when you have typed the `C-x C-a` command previously? This is why we ask you to give the exact sequence of characters you typed since starting to use Emacs.

You should not even say "visit a file" instead of `C-x C-f` unless you *know* that it makes no difference which visiting command is used. Similarly, rather than saying "if I have three characters on the line," say "after I type `RET A B C RET C-p`," if that is the way you entered the text.

If you are not in Fundamental mode when the problem occurs, you should say what mode you are in.

If the manifestation of the bug is an Emacs error message, it is important to report not just the text of the error message but a backtrace showing how the Lisp program in Emacs arrived at the error. To make the backtrace, you must execute the Lisp expression `(setq debug-on-error t)` before the error happens (that is to say, you must execute that expression and then make the bug happen). This causes the Lisp debugger to run (see Section 22.5 [Lisp Debug], page 173). The debugger's backtrace can be copied as text into the bug report. This use of the debugger is possible only if you know how to make the bug happen again. Do note the error message the first time the bug happens, so if you can't make it happen again, you can report at least that.

Check whether any programs you have loaded into the Lisp world, including your '`.emacs`' file, set any variables that may affect the functioning of Emacs. Also, see whether the problem happens in a freshly started Emacs without loading your '`.emacs`' file (start Emacs with the `-q` switch to prevent loading the init file.) If the problem does *not* occur then, it is essential that we know the contents of any programs that you must load into the Lisp world in order to cause the problem to occur.

If the problem does depend on an init file or other Lisp programs that are not part of the standard Emacs system, then you should make sure it is not a bug in those programs by complaining to their maintainers first. After they verify that they are using Emacs in a way that is supposed to work, they should report the bug.

If you can tell us a way to cause the problem without visiting any files, please do so. This makes it much easier to debug. If you do need files, make sure you arrange for us to see their exact contents. For example, it can often matter whether there are spaces at the ends of lines, or a newline after the last line in the buffer (nothing ought to care whether the last line is terminated, but tell that to the bugs).

The easy way to record the input to Emacs precisely is to to write a dribble file; execute the Lisp expression

```
(open-dribble-file "~/dribble")
```

using `Meta-ESC` or from the '`*scratch*`' buffer just after starting Emacs. From then on, all Emacs input will be written in the specified dribble file until the Emacs process is killed.

For possible display bugs, it is important to report the terminal type (the value of environment variable `TERM`), the complete termcap entry for the terminal from '`/etc/termcap`' (since that file is not identical on all machines), and the output that Emacs actually sent to the terminal. The way to collect this output is to execute the Lisp expression

```
(open-termscript "~/termscript")
```

using `Meta-ESC` or from the '`*scratch*`' buffer just after starting Emacs. From then on, all output from Emacs to the terminal will be written in the specified termscript file as well, until the Emacs process is killed. If the problem happens when Emacs starts up, put this expression into your '`.emacs`' file so that the termscript file will be open when Emacs displays the screen for the first time. Be warned: it is often difficult, and sometimes impossible, to fix a terminal-dependent bug without access to a terminal of the type that stimulates the bug.

The address for reporting bugs is

GNU Emacs Bugs
545 Tech Sq, rm 703
Cambridge, MA 02139

or send email to '`mit-eddie!bug-gnu-emacs`' (Usenet) or '`bug-gnu-emacs@prep.ai.mit.edu`' (Internet).

Once again, we do not promise to fix the bug; but if the bug is serious, or ugly, or easy to fix, chances are we will want to.

Emacs Version 17 Antinews

For those of you who are downgrading from version 18 to version 17 of GNU Emacs, here is a list of the old features. You will note many simplifications; complicated features have been eliminated. The list does not include changes affecting only topics not dealt with in this manual.

General Changes

- Vi, EDT and Gosmacs emulation have been eliminated in version 17. The idea is that other editors are obsolete and nobody should want to remember they exist.
- The buffer-sorting commands of version 18 have been eliminated in version 17. See Section 27.3 [Sorting], page 209.
- The `M-TAB` command for completion of Lisp symbol names has been eliminated.
- The `M-/` command for dynamic abbrev expansion has been eliminated.
- `C-M-v` is no longer redefined in the minibuffer. It has its standard meaning, which is to scroll the "next" window. In the minibuffer, the "next" window is always the one at the top of the screen. See Chapter 17 [Windows], page 111.
- The old command `M-x occur-menu` is now the way to ask for a list of matches for a regexp and then pick one and move point to it. Refer to its on-line documentation for full details of its use. `M-x occur` has been simplified and now just displays a list of matches with no fancy options. See Section 13.8 [Other Repeating Search], page 81.
- Incremental searches both ordinary and regexp now share a single default search string which is the last thing searched for by either kind of incremental search. They do not wrap to the beginning or end of the buffer. See Section 13.1.1 [Incremental Search], page 71.
- The variables `search-low-speed` and `search-slow-window-lines` have been renamed to start with `isearch` instead of `search`.
- Undo in version 17 clears the "modified" flag more often. If the buffer contents that result from undoing are the same as at a prior instant when the "modified" flag was clear, the "modified" flag is cleared again. See Chapter 5 [Undo], page 33.
- `C-x C-v` is allowed only when the current buffer is visiting a file. See Section 15.2 [Visiting], page 88.
- Auto-save file names in version 17 do not have a final '`#`'. The auto-save file name for a file '`foo.c`' is therefore '`#foo.c`'. See Section 15.5 [Auto Save], page 96.
- `M-x recover-file` works more simply. It does not compare the dates of the specified file and its auto-save file; it does not display a directory listing for them. You must figure out on your own whether you want to recover the file from its auto-save file.

- Some of the command line switches have been eliminated (see Section 3.2 [Command Switches], page 22). Switches eliminated include '-insert' and '-i', and the alternate names '-funcall', '-load', '-user' and '-no-init-file'.

Changes in Major Modes

- Fortran mode has been eliminated.
- Nroff mode no longer defines a syntax for comments (see Section 20.1.1 [Nroff Mode], page 122).
- The two kinds of TeX mode have been combined into one; `M-x tex-mode` simply turns on this mode instead of choosing among two others. A further simplification is the elimination of the commands `C-c C-f`, `C-c C-k`, `C-c C-l` and `C-c C-q`. See Section 20.1.2 [TeX Mode], page 122.
- All the special commands of Outline mode have been moved to new keys or eliminated (see Section 20.1.3 [Outline Mode], page 126).
  - `C-c C-n` becomes `M-}`.
  - `C-c C-p` becomes `M-{`.
  - `C-c C-f`, `C-c C-b` and `C-c C-u` are eliminated.

  The variable `outline-regexp` has also been eliminated in version 17.
- In C mode, `TAB` always reindents the current line. The variable `c-tab-always-indent` has been eliminated and Emacs acts as if it were `t`. See Section 21.4.4 [C Indent], page 146.
- Linefeed is now redefined in C mode so that it reindents (with `TAB`) both of the lines that result from breaking the current line.
- The special commands used in Picture mode to specify the direction of cursor motion after self-inserting characters have been given new keys (see Chapter 24 [Picture], page 183). They are now
  - `` M-` `` to request leftward motion.
  - `M-'` to request rightward motion.
  - `M--` to request upward motion.
  - `M-=` to request downward motion.
  - `` C-c ` `` to request upward and leftward motion.
  - `C-c '` to request upward and rightward motion.
  - `C-c /` to request downward and leftward motion.
  - `C-c \` to request downward and rightward motion.
- The special `C-c` commands of Mail mode have been given new keys (see Chapter 25 [Sending Mail], page 187).
  - `C-c C-f C-s` becomes `C-c s`

   - `C-c C-f C-t` becomes `C-c t`
   - `C-c C-f C-b` becomes `C-c b`
   - `C-c C-f C-c` becomes `C-c c`
   - `C-c C-y` becomes `C-c y`
   - `C-c C-q` becomes `C-c q`
   - `C-c C-w` becomes `C-c w`
- The `g` command in Dired has been removed (see Section 15.7 [Dired], page 99).

Init Files and Library Changes

- The commands `load-file` and `load-library` are replaced with one command, `load`. This command is logically the same as version 18 `load-library`, but due to changes in the order of searching it can also serve in place of `load-file`. See Section 22.3.1 [Loading], page 169.

  The search order in version 17 is:

  1. Search all the directories in the search path for the file name as given.
  2. Append the suffix '`.elc`' and search all the directories.
  3. Remove the final '`c`', resulting in a suffix '`.el`', and search all the directories.

  The search path in version 17 normally starts with `nil`, meaning the current default directory.

  As a result, the first file name that `load` tries is the one `load-file` would use in version 18: no suffix, and current default directory.
- The default init file is called '`default-profile`' instead of '`default.el`' or '`default.elc`'. Also, it is executed only if you have no init file of your own.
- The terminal-independent keypad support in the '`keypad`' library has been eliminated. See Chapter 29 [Terminal Init], page 237.
- The function `setq-default` has been eliminated. Use `set-default` and quote the variable name, as in

  (`set-default '`*variable value*)

  Several built-in variables now are always local to all buffers.

  These variables are `tab-width`, `ctl-arrow`, `truncate-lines`, `fill-column`, `left-margin`, `mode-line-format`, `abbrev-mode`, `overwrite-mode`, `case-fold-search`, `auto-fill-hook`, `selective-display`.

  `set-default` does not work with these variables. They do have defaults, but the defaults affect only buffers yet to be created. The only way to set the default for variable *foo* is to set the variable named `default-`*foo*, such as `default-case-fold-search` and `default-fill-column`.

- Some variables have been eliminated. Emacs version 17 always behaves as if they were `nil`.
  - `backup-by-copying-when-mismatch`
  - `find-file-hooks`
  - `find-file-not-found-hooks`
  - `write-file-hooks`
  - `file-precious-flag`
  - `no-redraw-on-reenter`

The GNU Manifesto

What's GNU? Gnu's Not Unix!

GNU, which stands for Gnu's Not Unix, is the name for the complete Unix-compatible software system which I am writing so that I can give it away free to everyone who can use it. Several other volunteers are helping me. Contributions of time, money, programs and equipment are greatly needed.

So far we have an Emacs text editor with Lisp for writing editor commands, a source level debugger, a yacc-compatible parser generator, a linker, and around 35 utilities. A shell (command interpreter) is nearly completed. A new portable optimizing C compiler has compiled itself and may be released this year. An initial kernel exists but many more features are needed to emulate Unix. When the kernel and compiler are finished, it will be possible to distribute a GNU system suitable for program development. We will use TeX as our text formatter, but an nroff is being worked on. We will use the free, portable X window system as well. After this we will add a portable Common Lisp, an Empire game, a spreadsheet, and hundreds of other things, plus on-line documentation. We hope to supply, eventually, everything useful that normally comes with a Unix system, and more.

GNU will be able to run Unix programs, but will not be identical to Unix. We will make all improvements that are convenient, based on our experience with other operating systems. In particular, we plan to have longer filenames, file version numbers, a crashproof file system, filename completion perhaps, terminal-independent display support, and perhaps eventually a Lisp-based window system through which several Lisp programs and ordinary Unix programs can share a screen. Both C and Lisp will be available as system programming languages. We will try to support UUCP, MIT Chaosnet, and Internet protocols for communication.

GNU is aimed initially at machines in the 68000/16000 class with virtual memory, because they are the easiest machines to make it run on. The extra effort to make it run on smaller machines will be left to someone who wants to use it on them.

To avoid horrible confusion, please pronounce the 'G' in the word 'GNU' when it is the name of this project.

Why I Must Write GNU

I consider that the golden rule requires that if I like a program I must share it with other people who like it. Software sellers want to divide the users and conquer them, making each user agree not to share with others. I refuse to break solidarity with other users in this way. I cannot in good conscience sign a nondisclosure agreement or a software license agreement. For years I worked within the Artificial Intelligence Lab to resist such tendencies and other inhospitalities, but eventually they had gone too far: I could not remain in an institution where such things are done for me against my will.

So that I can continue to use computers without dishonor, I have decided to put together a sufficient body of free software so that I will be able to get along without any software that is not free. I have resigned from the AI lab to deny MIT any legal excuse to prevent me from giving GNU away.

Why GNU Will Be Compatible with Unix

Unix is not my ideal system, but it is not too bad. The essential features of Unix seem to be good ones, and I think I can fill in what Unix lacks without spoiling them. And a system compatible with Unix would be convenient for many other people to adopt.

How GNU Will Be Available

GNU is not in the public domain. Everyone will be permitted to modify and redistribute GNU, but no distributor will be allowed to restrict its further redistribution. That is to say, proprietary modifications will not be allowed. I want to make sure that all versions of GNU remain free.

Why Many Other Programmers Want to Help

I have found many other programmers who are excited about GNU and want to help.

Many programmers are unhappy about the commercialization of system software. It may enable them to make more money, but it requires them to feel in conflict with other programmers in general rather than feel as comrades. The fundamental act of friendship among programmers is the sharing of programs; marketing arrangements now typically used essentially forbid programmers to treat others as friends. The purchaser of software must choose between friendship and obeying the law. Naturally, many decide that friendship is more important. But those who believe in law often do not feel at ease with either choice. They become cynical and think that programming is just a way of making money.

By working on and using GNU rather than proprietary programs, we can be hospitable to everyone and obey the law. In addition, GNU serves as an example to inspire and a banner to rally others to join us in sharing. This can give us a feeling of harmony which is impossible if we use software that is not free. For about half the programmers I talk to, this is an important happiness that money cannot replace.

How You Can Contribute

I am asking computer manufacturers for donations of machines and money. I'm asking individuals for donations of programs and work.

One consequence you can expect if you donate machines is that GNU will run on them at an early date. The machines should be complete, ready to use systems, approved for use in a residential area, and not in need of sophisticated cooling or power.

I have found very many programmers eager to contribute part-time work for GNU. For most projects, such part-time distributed work would be very hard to coordinate; the independently-written parts would not work together. But for the particular task of replacing Unix, this problem is absent. A complete Unix system contains hundreds of utility programs, each of which is documented separately. Most interface specifications are fixed by Unix compatibility. If each contributor can write a compatible replacement for a single Unix utility, and make it work properly in place of the original on a Unix system, then these utilities will work right when put together. Even allowing for Murphy to create a few unexpected problems, assembling these components will be a feasible task. (The kernel will require closer communication and will be worked on by a small, tight group.)

If I get donations of money, I may be able to hire a few people full or part time. The salary won't be high by programmers' standards, but I'm looking for people for whom building community spirit is as important as making money. I view this as a way of enabling dedicated people to devote their full energies to working on GNU by sparing them the need to make a living in another way.

Why All Computer Users Will Benefit

Once GNU is written, everyone will be able to obtain good system software free, just like air.

This means much more than just saving everyone the price of a Unix license. It means that much wasteful duplication of system programming effort will be avoided. This effort can go instead into advancing the state of the art.

Complete system sources will be available to everyone. As a result, a user who needs changes in the system will always be free to make them himself, or hire any available programmer or company to make them for him. Users will no longer be at the mercy of one programmer or company which owns the sources and is in sole position to make changes.

Schools will be able to provide a much more educational environment by encouraging all students to study and improve the system code. Harvard's computer lab used to have the policy that no program could be installed on the system if its sources were not on public display, and upheld it by actually refusing to install certain programs. I was very much inspired by this.

Finally, the overhead of considering who owns the system software and what one is or is not entitled to do with it will be lifted.

Arrangements to make people pay for using a program, including licensing of copies, always incur a tremendous cost to society through the cumbersome mechanisms necessary to figure out how much (that is, which programs) a person must pay for. And only a police state can force everyone to obey them. Consider a space station where air must be manufactured at great cost: charging each breather per liter of air may be fair, but wearing the metered gas mask all day and all night is intolerable even if everyone can afford to pay the air bill. And the TV cameras everywhere to see if you ever take the mask off are outrageous. It's better to support the air plant with a head tax and chuck the masks.

Copying all or parts of a program is as natural to a programmer as breathing, and as productive. It ought to be as free.

Some Easily Rebutted Objections to GNU's Goals

"Nobody will use it if it is free, because that means they can't rely on any support."

"You have to charge for the program to pay for providing the support."

If people would rather pay for GNU plus service than get GNU free without service, a company to provide just service to people who have obtained GNU free ought to be profitable.

We must distinguish between support in the form of real programming work and mere hand-holding. The former is something one cannot rely on from a software vendor. If your problem is not shared by enough people, the vendor will tell you to get lost.

If your business needs to be able to rely on support, the only way is to have all the necessary sources and tools. Then you can hire any available person to fix your problem; you are not at the mercy of any individual. With Unix, the price of sources puts this out of consideration for most businesses. With GNU this will be easy. It is still possible for there to be no available competent person, but this problem cannot be blamed on distibution arrangements. GNU does not eliminate all the world's problems, only some of them.

Meanwhile, the users who know nothing about computers need handholding: doing things for them which they could easily do themselves but don't know how.

Such services could be provided by companies that sell just hand-holding and repair service. If it is true that users would rather spend money and get a product with service, they will also be willing to buy the service having got the product free. The service companies will compete in quality and price; users will not be tied to any particular one. Meanwhile, those of us who don't need the service should be able to use the program without paying for the service.

> "You cannot reach many people without advertising, and you must charge for the program to support that."

> "It's no use advertising a program people can get free."

There are various forms of free or very cheap publicity that can be used to inform numbers of computer users about something like GNU. But it may be true that one can reach more microcomputer users with advertising. If this is really so, a business which advertises the service of copying and mailing GNU for a fee ought to be successful enough to pay for its advertising and more. This way, only the users who benefit from the advertising pay for it.

On the other hand, if many people get GNU from their friends, and such companies don't succeed, this will show that advertising was not really necessary to spread GNU. Why is it that free market advocates don't want to let the free market decide this?

> "My company needs a proprietary operating system to get a competitive edge."

GNU will remove operating system software from the realm of competition. You will not be able to get an edge in this area, but neither will your competitors be able to get an edge over you. You and they will compete in other areas, while benefitting mutually in this one. If your business is selling an operating system, you will not like GNU, but that's tough on you. If your business is something else, GNU can save you from being pushed into the expensive business of selling operating systems.

I would like to see GNU development supported by gifts from many manufacturers and users, reducing the cost to each.

> "Don't programmers deserve a reward for their creativity?"

If anything deserves a reward, it is social contribution. Creativity can be a social contribution, but only in so far as society is free to use the results. If programmers deserve to be rewarded for creating innovative programs, by the same token they deserve to be punished if they restrict the use of these programs.

> "Shouldn't a programmer be able to ask for a reward for his creativity?"

There is nothing wrong with wanting pay for work, or seeking to maximize one's income, as long as one does not use means that are destructive. But the means customary in the field of software today are based on destruction.

Extracting money from users of a program by restricting their use of it is destructive because the restrictions reduce the amount and the ways that the program can be used. This reduces the amount of wealth that humanity derives from the program. When there is a deliberate choice to restrict, the harmful consequences are deliberate destruction.

The reason a good citizen does not use such destructive means to become wealthier is that, if everyone did so, we would all become poorer from the mutual destructiveness. This is Kantian ethics; or, the Golden Rule. Since I do not like the consequences that result if everyone hoards information, I am required to consider it wrong for one to do so. Specifically, the desire to be rewarded for one's creativity does not justify depriving the world in general of all or part of that creativity.

> "Won't programmers starve?"

I could answer that nobody is forced to be a programmer. Most of us cannot manage to get any money for standing on the street and making faces. But we are not, as a result, condemned to spend our lives standing on the street making faces, and starving. We do something else.

But that is the wrong answer because it accepts the questioner's implicit assumption: that without ownership of software, programmers cannot possibly be paid a cent. Supposedly it is all or nothing.

The real reason programmers will not starve is that it will still be possible for them to get paid for programming; just not paid as much as now.

Restricting copying is not the only basis for business in software. It is the most common basis because it brings in the most money. If it were prohibited, or rejected by the customer, software business would move to other bases of organization which are now used less often. There are always numerous ways to organize any kind of business.

Probably programming will not be as lucrative on the new basis as it is now. But that is not an argument against the change. It is not considered an injustice that sales clerks make the salaries that they now do. If programmers made the same, that would not be an injustice either. (In practice they would still make considerably more than that.)

"Don't people have a right to control how their creativity is used?"

"Control over the use of one's ideas" really constitutes control over other people's lives; and it is usually used to make their lives more difficult.

People who have studied the issue of intellectual property rights carefully (such as lawyers) say that there is no intrinsic right to intellectual property. The kinds of supposed intellectual property rights that the government recognizes were created by specific acts of legislation for specific purposes.

For example, the patent system was established to encourage inventors to disclose the details of their inventions. Its purpose was to help society rather than to help inventors. At the time, the life span of 17 years for a patent was short compared with the rate of advance of the state of the art. Since patents are an issue only among manufacturers, for whom the cost and effort of a license agreement are small compared with setting up production, the patents often do not do much harm. They do not obstruct most individuals who use patented products.

The idea of copyright did not exist in ancient times, when authors frequently copied other authors at length in works of non-fiction. This practice was useful, and is the only way many authors' works have survived even in part. The copyright system was created expressly for the purpose of encouraging authorship. In the domain for which it was invented—books, which could be copied economically only on a printing press—it did little harm, and did not obstruct most of the individuals who read the books.

All intellectual property rights are just licenses granted by society because it was thought, rightly or wrongly, that society as a whole would benefit by granting them. But in any particular situation, we have to ask: are we really better off granting such license? What kind of act are we licensing a person to do?

The case of programs today is very different from that of books a hundred years ago. The fact that the easiest way to copy a program is from one neighbor to another, the fact that a program has both source code and object code which are distinct, and the fact that a program is used rather than read and enjoyed, combine to create a situation in which a person who enforces a copyright is harming society as a whole both materially and spiritually; in which a person should not do so regardless of whether the law enables him to.

"Competition makes things get done better."

The paradigm of competition is a race: by rewarding the winner, we encourage everyone to run faster. When capitalism really works this way, it does a good job; but its defenders are wrong in assuming it always works this way. If the runners forget why the reward is offered and become intent on winning, no matter how, they may find other strategies—such as, attacking other runners. If the runners get into a fist fight, they will all finish late.

Proprietary and secret software is the moral equivalent of runners in a fist fight. Sad to say, the only referee we've got does not seem to object to fights; he just regulates them ("For every ten yards you run, you can fire one shot"). He really ought to break them up, and penalize runners for even trying to fight.

"Won't everyone stop programming without a monetary incentive?"

Actually, many people will program with absolutely no monetary incentive. Programming has an irresistible fascination for some people, usually the people who are best at it. There is no shortage of professional musicians who keep at it even though they have no hope of making a living that way.

But really this question, though commonly asked, is not appropriate to the situation. Pay for programmers will not disappear, only become less. So the right question is, will anyone program with a reduced monetary incentive? My experience shows that they will.

For more than ten years, many of the world's best programmers worked at the Artificial Intelligence Lab for far less money than they could have had anywhere else. They got many kinds of non-monetary rewards: fame and appreciation, for example. And creativity is also fun, a reward in itself.

Then most of them left when offered a chance to do the same interesting work for a lot of money.

What the facts show is that people will program for reasons other than riches; but if given a chance to make a lot of money as well, they will come to expect and demand it. Low-paying organizations do poorly in competition with high-paying ones, but they do not have to do badly if the high-paying ones are banned.

> "We need the programmers desperately. If they demand that we stop helping our neighbors, we have to obey."

You're never so desperate that you have to obey this sort of demand. Remember: millions for defense, but not a cent for tribute!

> "Programmers need to make a living somehow."

In the short run, this is true. However, there are plenty of ways that programmers could make a living without selling the right to use a program. This way is customary now because it brings programmers and businessmen the most money, not because it is the only way to make a living. It is easy to find other ways if you want to find them. Here are a number of examples.

A manufacturer introducing a new computer will pay for the porting of operating systems onto the new hardware.

The sale of teaching, hand-holding and maintenance services could also employ programmers.

People with new ideas could distribute programs as freeware, asking for donations from satisfied users, or selling hand-holding services. I have met people who are already working this way successfully.

Users with related needs can form users' groups, and pay dues. A group would contract with programming companies to write programs that the group's members would like to use.

All sorts of development can be funded with a Software Tax:

> Suppose everyone who buys a computer has to pay x percent of the price as a software tax. The government gives this to an agency like the NSF to spend on software development.
>
> But if the computer buyer makes a donation to software development himself, he can take a credit against the tax. He can donate to the project of his own choosing—often, chosen because he hopes to use the results when it is done. He can take a credit for any amount of donation up to the total tax he had to pay.
>
> The total tax rate could be decided by a vote of the payers of the tax, weighted according to the amount they will be taxed on.
>
> The consequences:
>
> - The computer-using community supports software development.
> - This community decides what level of support is needed.
> - Users who care which projects their share is spent on can choose this for themselves.

In the long run, making programs free is a step toward the post-scarcity world, where nobody will have to work very hard just to make a living. People will be free to devote themselves to activities that are fun, such as programming, after spending the necessary ten hours a week on required tasks such as legislation, family counseling, robot repair and asteroid prospecting. There will be no need to be able to make a living from programming.

We have already greatly reduced the amount of work that the whole society must do for its actual productivity, but only a little of this has translated itself into leisure for workers because much nonproductive activity is required to accompany productive activity. The main causes of this are bureaucracy and isometric struggles against competition. Free software will greatly reduce these drains in the area of software production. We must do this, in order for technical gains in productivity to translate into less work for us.

Glossary

Abbrev
: An abbrev is a text string which expands into a different text string when present in the buffer. For example, you might define a short word as an abbrev for a long phrase that you want to insert frequently. See Chapter 23 [Abbrevs], page 177.

Aborting
: Aborting means getting out of a recursive edit (q.v.). The commands `C-]` and `M-x top-level` are used for this. See Section 29.1 [Quitting], page 237.

Auto Fill mode
: Auto Fill mode is a minor mode in which text that you insert is automatically broken into lines of fixed width. See Section 20.6 [Filling], page 134.

Auto Saving
: Auto saving is when Emacs automatically stores the contents of an Emacs buffer in a specially-named file so that the information will not be lost if the buffer is lost due to a system error or user error. See Section 15.5 [Auto Save], page 96.

Backup File
: A backup file records the contents that a file had before the current editing session. Emacs makes backup files automatically to help you track down or cancel changes you later regret making. See Section 15.3.1 [Backup], page 92.

Balance Parentheses
: Emacs can balance parentheses manually or automatically. Manual balancing is done by the commands to move over balanced expressions (see Section 21.2 [Lists], page 140). Automatic balancing is done by blinking the parenthesis that matches one just inserted (see Section 21.5 [Matching Parens], page 149).

Bind
: To bind a key is to change its binding (q.v.). See Section 28.4.2 [Rebinding], page 228.

Binding
: A key gets its meaning in Emacs by having a binding which is a command (q.v.), a Lisp function that is run when the key is typed. See Section 2.3 [Commands], page 19. Customization often involves rebinding a character to a different command function. The bindings of all keys are recorded in the keymaps (q.v.). See Section 28.4.1 [Keymaps], page 226.

Blank Lines
: Blank lines are lines that contain only whitespace. Emacs has several commands for operating on the blank lines in the buffer.

Buffer
: The buffer is the basic editing unit; one buffer corresponds to one piece of text being edited. You can have several buffers, but at any time you are editing only one, the 'selected' buffer, though several can be visible when you are using multiple windows. See Chapter 16 [Buffers], page 105.

Buffer Selection History
: Emacs keeps a buffer selection history which records how recently each Emacs buffer

has been selected. This is used for choosing a buffer to select. See Chapter 16 [Buffers], page 105.

C- `C' in the name of a character is an abbreviation for Control. See Section 2.1 [Characters], page 17.

C-M- `C-M-' in the name of a character is an abbreviation for Control-Meta. See Section 2.1 [Characters], page 17.

Case Conversion
: Case conversion means changing text from upper case to lower case or vice versa. See Section 20.7 [Case], page 137, for the commands for case conversion.

Characters
: Characters form the contents of an Emacs buffer; also, Emacs commands are invoked by keys (q.v.), which are sequences of one or more characters. See Section 2.1 [Characters], page 17.

Command A command is a Lisp function specially defined to be able to serve as a key binding in Emacs. When you type a key (q.v.), its binding (q.v.) is looked up in the relevant keymaps (q.v.) to find the command to run. See Section 2.3 [Commands], page 19.

Command Name
: A command name is the name of a Lisp symbol which is a command (see Section 2.3 [Commands], page 19). You can invoke any command by its name using `M-x` (see Chapter 7 [M-x], page 43).

Comments
: A comment is text in a program which is intended only for humans reading the program, and is marked specially so that it will be ignored when the program is loaded or compiled. Emacs offers special commands for creating, aligning and killing comments. See Section 21.6.2 [Comments], page 152.

Compilation
: Compilation is the process of creating an executable program from source code. Emacs has commands for compiling files of Emacs Lisp code (see Section 22.3 [Lisp Libraries], page 169) and programs in C and other languages (see Section 22.1 [Compilation], page 167).

Complete Key
: A complete key is a character or sequence of characters which, when typed by the user, fully specifies one action to be performed by Emacs. For example, `X` and `Control-f` and `Control-x m` are keys. Keys derive their meanings from being bound (q.v.) to commands (q.v.). Thus, `X` is conventionally bound to a command to insert `X' in the buffer; `C-x m` is conventionally bound to a command to begin composing a mail message. See Section 2.2 [Keys], page 18.

Completion
: Completion is what Emacs does when it automatically fills out an abbreviation for a

name into the entire name. Completion is done for minibuffer (q.v.) arguments when the set of possible valid inputs is known; for example, on command names, buffer names, and file names. Completion occurs when `TAB`, `SPC` or `RET` is typed. See Section 6.3.2 [Completion], page 38.

Continuation Line
: When a line of text is longer than the width of the screen, it takes up more than one screen line when displayed. We say that the text line is continued, and all screen lines used for it after the first are called continuation lines. See Section 4.5 [Basic Editing], page 28.

Control-Character
: ASCII characters with octal codes 0 through 037, and also code 0177, do not have graphic images assigned to them. These are the control characters. Any control character can be typed by holding down the `CTRL` key and typing some other character; some have special keys on the keyboard. `RET`, `TAB`, `ESC`, `LFD` and `DEL` are all control characters. See Section 2.1 [Characters], page 17.

Copyleft
: A copyleft is a notice giving the public legal permission to redistribute a program or other work of art. Copylefts are used by leftists to enrich the public just as copyrights are used by rightists to gain power over the public.

Current Buffer
: The current buffer in Emacs is the Emacs buffer on which most editing commands operate. You can select any Emacs buffer as the current one. See Chapter 16 [Buffers], page 105.

Current Line
: The line point is on (see Section 1.1 [Point], page 13).

Current Paragraph
: The paragraph that point is in. If point is between paragraphs, the current paragraph is the one that follows point. See Section 20.4 [Paragraphs], page 132.

Current Defun
: The defun (q.v.) that point is in. If point is between defuns, the current defun is the one that follows point. See Section 21.3 [Defuns], page 142.

Cursor
: The cursor is the rectangle on the screen which indicates the position called point (q.v.) at which insertion and deletion takes place. The cursor is on or under the character that follows point. Often people speak of 'the cursor' when, strictly speaking, they mean 'point'. See Section 4.5 [Basic Editing], page 28.

Customization
: Customization is making minor changes in the way Emacs works. It is often done by setting variables (see Section 28.2 [Variables], page 218) or by rebinding keys (see Section 28.4.1 [Keymaps], page 226).

Default Argument

The default for an argument is the value that will be assumed if you do not specify one. When the minibuffer is used to read an argument, the default argument is used if you just type `RET`. See Chapter 6 [Minibuffer], page 35.

Default Directory
: When you specify a file name that does not start with '`/`' or '`~`', it is interpreted relative to the current buffer's default directory. See Section 6.1 [Minibuffer File], page 35.

Defun
: A defun is a list at the top level of parenthesis or bracket structure in a program. It is so named because most such lists in Lisp programs are calls to the Lisp function `defun`. See Section 21.3 [Defuns], page 142.

`DEL`
: `DEL` is a character that runs the command to delete one character of text. See Section 4.5 [Basic Editing], page 28.

Deletion
: Deletion means erasing text without saving it. Emacs deletes text only when it is expected not to be worth saving (all whitespace, or only one character). The alternative is killing (q.v.). See Section 10.1.3 [Killing], page 55.

Deletion of Files
: Deleting a file means erasing it from the file system. See Section 15.8 [Misc File Ops], page 102.

Deletion of Messages
: Deleting a message means flagging it to be eliminated from your mail file. This can be undone by undeletion until the mail file is expunged. See Section 26.3 [Rmail Deletion], page 195.

Deletion of Windows
: Deleting a window means eliminating it from the screen. Other windows expand to use up the space. The deleted window can never come back, but no actual text is thereby lost. See Chapter 17 [Windows], page 111.

Directory
: Files in the Unix file system are grouped into file directories. See Section 15.6 [Directories], page 98.

Dired
: Dired is the Emacs facility that displays the contents of a file directory and allows you to "edit the directory", performing operations on the files in the directory. See Section 15.7 [Dired], page 99.

Disabled Command
: A disabled command is one that you may not run without special confirmation. The usual reason for disabling a command is that it is confusing for beginning users. See Section 28.4.3 [Disabling], page 229.

Dribble File
: A file into which Emacs writes all the characters that the user types on the keyboard. Dribble files are used to make a record for debugging Emacs bugs. Emacs does not make a dribble file unless you tell it to. See [Bugs], page 247.

Echo Area The echo area is the bottom line of the screen, used for echoing the arguments to commands, for asking questions, and printing brief messages (including error messages). See Section 1.2 [Echo Area], page 14.

Echoing Echoing is acknowledging the receipt of commands by displaying them (in the echo area). Emacs never echoes single-character keys; longer keys echo only if you pause while typing them.

Error An error occurs when an Emacs command cannot execute in the current circumstances. When an error occurs, execution of the command stops (unless the command has been programmed to do otherwise) and Emacs reports the error by printing an error message (q.v.). Type-ahead is discarded. Then Emacs is ready to read another editing command.

Error Messages
: Error messages are single lines of output printed by Emacs when the user asks for something impossible to do (such as, killing text forward when point is at the end of the buffer). They appear in the echo area, accompanied by a beep.

`ESC` `ESC` is a character, used to end incremental searches and as a prefix for typing Meta characters on keyboards lacking a `META` key. Unlike the `META` key (which, like the `SHIFT` key, is held down while another character is typed), the `ESC` key is pressed once and applies to the next character typed.

Fill Prefix The fill prefix is a string that should be expected at the beginning of each line when filling is done. It is not regarded as part of the text to be filled. See Section 20.6 [Filling], page 134.

Filling Filling text means moving text from line to line so that all the lines are approximately the same length. See Section 20.6 [Filling], page 134.

Global Global means 'independent of the current environment; in effect throughout Emacs'. It is the opposite of local (q.v.). Particular examples of the use of 'global' appear below.

Global Abbrev
: A global definition of an abbrev (q.v.) is effective in all major modes that do not have local (q.v.) definitions for the same abbrev. See Chapter 23 [Abbrevs], page 177.

Global Keymap
: The global keymap (q.v.) contains key bindings that are in effect except when overridden by local key bindings in a major mode's local keymap (q.v.). See Section 28.4.1 [Keymaps], page 226.

Global Substitution
: Global substitution means replacing each occurrence of one string by another string through a large amount of text. See Section 13.7 [Replace], page 78.

Global Variable
: The global value of a variable (q.v.) takes effect in all buffers that do not have their own local (q.v.) values for the variable. See Section 28.2 [Variables], page 218.

Graphic Character
: Graphic characters are those assigned pictorial images rather than just names. All the non-Meta (q.v.) characters except for the Control (q.v.) characters are graphic characters. These include letters, digits, punctuation, and spaces; they do not include `RET` or `ESC`. In Emacs, typing a graphic character inserts that character (in ordinary editing modes). See Section 4.5 [Basic Editing], page 28.

Grinding
: Grinding means adjusting the indentation in a program to fit the nesting structure. See Chapter 19 [Indentation], page 117.

Hardcopy
: Hardcopy means printed output. Emacs has commands for making printed listings of text in Emacs buffers. See Section 27.5 [Hardcopy], page 213.

`HELP`
: You can type `HELP` at any time to ask what options you have, or to ask what any command does. `HELP` is really `Control-h`. See Section 8.4 [Help], page 48.

Inbox
: An inbox is a file in which mail is delivered by the operating system. Rmail transfers mail from inboxes to mail files (q.v.) in which the mail is then stored permanently or until explicitly deleted. See Section 26.4 [Rmail Inbox], page 197.

Indentation
: Indentation means blank space at the beginning of a line. Most programming languages have conventions for using indentation to illuminate the structure of the program, and Emacs has special features to help you set up the correct indentation. See Chapter 19 [Indentation], page 117.

Insertion
: Insertion means copying text into the buffer, either from the keyboard or from some other place in Emacs.

Justification
: Justification means adding extra spaces to lines of text to make them come exactly to a specified width. See Section 20.6 [Filling], page 134.

Keyboard Macros
: Keyboard macros are a way of defining new Emacs commands from sequences of existing ones, with no need to write a Lisp program. See Section 28.3 [Keyboard Macros], page 223.

Key
: A key is a sequence of characters that, when input to Emacs, specify or begin to specify a single action for Emacs to perform. That is, the sequence is not more than a single unit. If the key is enough to specify one action, it is a complete key (q.v.); if it is less than enough, it is a prefix key (q.v.). See Section 2.2 [Keys], page 18.

Keymap
: The keymap is the data structure that records the bindings (q.v.) of keys to the commands that they run. For example, the keymap binds the character `C-n` to the command function `next-line`. See Section 28.4.1 [Keymaps], page 226.

Kill Ring
: The kill ring is where all text you have killed recently is saved. You can reinsert any of the killed text still in the ring; this is called yanking (q.v.). See Section 10.2 [Yanking], page 55.

Killing — Killing means erasing text and saving it on the kill ring so it can be yanked (q.v.) later. Some other systems call this "cutting". Most Emacs commands to erase text do killing, as opposed to deletion (q.v.). See Section 10.1.3 [Killing], page 55.

Killing Jobs — Killing a job (such as, an invocation of Emacs) means making it cease to exist. Any data within it, if not saved in a file, is lost. See Section 3.1 [Exiting], page 21.

List — A list is, approximately, a text string beginning with an open parenthesis and ending with the matching close parenthesis. In C mode and other non-Lisp modes, groupings surrounded by other kinds of matched delimiters appropriate to the language, such as braces, are also considered lists. Emacs has special commands for many operations on lists. See Section 21.2 [Lists], page 140.

Local — Local means 'in effect only in a particular context'; the relevant kind of context is a particular function execution, a particular buffer, or a particular major mode. It is the opposite of 'global' (q.v.). Specific uses of 'local' in Emacs terminology appear below.

Local Abbrev — A local abbrev definition is effective only if a particular major mode is selected. In that major mode, it overrides any global definition for the same abbrev. See Chapter 23 [Abbrevs], page 177.

Local Keymap — A local keymap is used in a particular major mode; the key bindings (q.v.) in the current local keymap override global bindings of the same keys. See Section 28.4.1 [Keymaps], page 226.

Local Variable — A local value of a variable (q.v.) applies to only one buffer. See Section 28.2.3 [Locals], page 220.

M- — `M-` in the name of a character is an abbreviation for `META`, one of the modifier keys that can accompany any character. See Section 2.1 [Characters], page 17.

M-C- — '`M-C-`' in the name of a character is an abbreviation for Control-Meta; it means the same thing as '`C-M-`'. If your terminal lacks a real `META` key, you type a Control-Meta character by typing `ESC` and then typing the corresponding Control character. See Section 2.1 [Characters], page 17.

M-x — `M-x` is the key which is used to call an Emacs command by name. This is how commands that are not bound to keys are called. See Chapter 7 [M-x], page 43.

Mail — Mail means messages sent from one user to another through the computer system, to be read at the recipient's convenience. Emacs has commands for composing and sending mail, and for reading and editing the mail you have received. See Chapter 25 [Sending Mail], page 187. See Chapter 26 [Rmail], page 193, for how to read mail.

Mail File — A mail file is a file which is edited using Rmail and in which Rmail stores mail. See Chapter 26 [Rmail], page 193.

Major Mode
: The major modes are a mutually exclusive set of options each of which configures Emacs for editing a certain sort of text. Ideally, each programming language has its own major mode. See Chapter 18 [Major Modes], page 115.

Mark
: The mark points to a position in the text. It specifies one end of the region (q.v.), point being the other end. Many commands operate on all the text from point to the mark. See Chapter 9 [Mark], page 49.

Mark Ring
: The mark ring is used to hold several recent previous locations of the mark, just in case you want to move back to them. See Chapter 10 [Mark Ring], page 53.

Message
: See 'mail'.

Meta
: Meta is the name of a modifier bit which a command character may have. It is present in a character if the character is typed with the `META` key held down. Such characters are given names that start with `Meta-`. For example, `Meta-<` is typed by holding down `META` and at the same time typing `<` (which itself is done, on most terminals, by holding down `SHIFT` and typing `,`). See Section 2.1 [Characters], page 17.

Meta Character
: A Meta character is one whose character code includes the Meta bit.

Minibuffer
: The minibuffer is the window that appears when necessary inside the echo area (q.v.), used for reading arguments to commands. See Chapter 6 [Minibuffer], page 35.

Minor Mode
: A minor mode is an optional feature of Emacs which can be switched on or off independently of all other features. Each minor mode has a command to turn it on or off. See Section 28.1 [Minor Modes], page 217.

Mode Line
: The mode line is the line at the bottom of each text window (q.v.), which gives status information on the buffer displayed in that window. See Chapter 2 [Mode Line], page 17.

Modified Buffer
: A buffer (q.v.) is modified if its text has been changed since the last time the buffer was saved (or since when it was created, if it has never been saved). See Section 15.3 [Saving], page 90.

Moving Text
: Moving text means erasing it from one place and inserting it in another. This is done by killing (q.v.) and then yanking (q.v.). See Section 10.1.3 [Killing], page 55.

Named Mark
: A named mark is a register (q.v.) in its role of recording a location in text so that you can move point to that location. See Chapter 11 [Registers], page 63.

Narrowing
: Narrowing means creating a restriction (q.v.) that limits editing in the current buffer to only a part of the text in the buffer. Text outside that part is inaccessible to the user until the boundaries are widened again, but it is still there, and saving the file saves it all. See Section 27.2 [Narrowing], page 208.

Newline
: `LFD` characters in the buffer terminate lines of text and are called newlines. See Section 2.1 [Characters], page 17.

Numeric Argument
: A numeric argument is a number, specified before a command, to change the effect of the command. Often the numeric argument serves as a repeat count. See Section 4.9 [Arguments], page 31.

Option
: An option is a variable (q.v.) that exists so that you can customize Emacs by giving it a new value. See Section 28.2 [Variables], page 218.

Overwrite Mode
: Overwrite mode is a minor mode. When it is enabled, ordinary text characters replace the existing text after point rather than pushing it to the right. See Section 28.1 [Minor Modes], page 217.

Page
: A page is a unit of text, delimited by formfeed characters (ASCII Control-L, code 014) coming at the beginning of a line. Some Emacs commands are provided for moving over and operating on pages. See Section 20.5 [Pages], page 133.

Paragraphs
: Paragraphs are the medium-size unit of English text. There are special Emacs commands for moving over and operating on paragraphs. See Section 20.4 [Paragraphs], page 132.

Parsing
: We say that Emacs parses words or expressions in the text being edited. Really, all it knows how to do is find the other end of a word or expression. See Section 28.5 [Syntax], page 229.

Point
: Point is the place in the buffer at which insertion and deletion occur. Point is considered to be between two characters, not at one character. The terminal's cursor (q.v.) indicates the location of point. See Section 4.5 [Basic], page 28.

Prefix Key
: A prefix key is a key (q.v.) whose sole function is to introduce a set of multi-character keys. `Control-x` is an example of prefix key; thus, any two-character sequence starting with `C-x` is also a legitimate key. See Section 2.2 [Keys], page 18.

Primary Mail File
: Your primary mail file is the file named '`RMAIL`' in your home directory, where all mail that you receive is stored by Rmail unless you make arrangements to do otherwise. See Chapter 26 [Rmail], page 193.

Prompt
: A prompt is text printed to ask the user for input. Printing a prompt is called prompting. Emacs prompts always appear in the echo area (q.v.). One kind of prompting

happens when the minibuffer is used to read an argument (see Chapter 6 [Minibuffer], page 35); the echoing which happens when you pause in the middle of typing a multicharacter key is also a kind of prompting (see Section 1.2 [Echo Area], page 14).

Quitting
: Quitting means cancelling a partially typed command or a running command, using `C-g`. See Section 29.1 [Quitting], page 237.

Quoting
: Quoting means depriving a character of its usual special significance. In Emacs this is usually done with `Control-q`. What constitutes special significance depends on the context and on convention. For example, an "ordinary" character as an Emacs command inserts itself; so in this context, a special character is any character that does not normally insert itself (such as `DEL`, for example), and quoting it makes it insert itself as if it were not special. Not all contexts allow quoting. See Section 4.5 [Basic Editing], page 28.

Read-only Buffer
: A read-only buffer is one whose text you are not allowed to change. Normally Emacs makes buffers read-only when they contain text which has a special significance to Emacs; for example, Dired buffers. Visiting a file that is write protected also makes a read-only buffer. See Chapter 16 [Buffers], page 105.

Recursive Editing Level
: A recursive editing level is a state in which part of the execution of a command involves asking the user to edit some text. This text may or may not be the same as the text to which the command was applied. The mode line indicates recursive editing levels with square brackets ('`[`' and '`]`'). See Section 27.1 [Recursive Edit], page 207.

Redisplay
: Redisplay is the process of correcting the image on the screen to correspond to changes that have been made in the text being edited. See Chapter 1 [Screen], page 13.

Regexp
: See 'regular expression'.

Region
: The region is the text between point (q.v.) and the mark (q.v.). Many commands operate on the text of the region. See Chapter 9 [Mark], page 49.

Registers
: Registers are named slots in which text or buffer positions or rectangles can be saved for later use. See Chapter 11 [Registers], page 63.

Regular Expression
: A regular expression is a pattern that can match various text strings; for example, '`l[0-9]+`' matches '`l`' followed by one or more digits. See Section 13.5 [Regexps], page 74.

Replacement
: See 'global substitution'.

Restriction
: A buffer's restriction is the amount of text, at the beginning or the end of the buffer, that is temporarily invisible and inaccessible. Giving a buffer a nonzero amount of restriction is called narrowing (q.v.). See Section 27.2 [Narrowing], page 208.

RET
: RET is a character than in Emacs runs the command to insert a newline into the text. It is also used to terminate most arguments read in the minibuffer (q.v.). See Section 2.1 [Characters], page 17.

Saving
: Saving a buffer means copying its text into the file that was visited (q.v.) in that buffer. This is the way text in files actually gets changed by your Emacs editing. See Section 15.3 [Saving], page 90.

Scrolling
: Scrolling means shifting the text in the Emacs window so as to see a different part of the buffer. See Chapter 12 [Display], page 65.

Searching
: Searching means moving point to the next occurrence of a specified string. See Chapter 13 [Search], page 69.

Selecting
: Selecting a buffer means making it the current (q.v.) buffer. See Chapter 16 [Buffers], page 105.

Self-documentation
: Self-documentation is the feature of Emacs which can tell you what any command does, or give you a list of all commands related to a topic you specify. You ask for self-documentation with the help character, `C-h`. See Section 8.4 [Help], page 48.

Sentences
: Emacs has commands for moving by or killing by sentences. See Section 20.3 [Sentences], page 131.

Sexp
: A sexp (short for 's-expression') is the basic syntactic unit of Lisp in its textual form: either a list, or Lisp atom. Many Emacs commands operate on sexps. The term 'sexp' is generalized to languages other than Lisp, to mean a syntactically recognizable expression. See Section 21.2 [Lists], page 140.

Simultaneous Editing
: Simultaneous editing means two users modifying the same file at once. Simultaneous editing if not detected can cause one user to lose his work. Emacs detects all cases of simultaneous editing and warns the user to investigate them. See Section 15.3.2 [Simultaneous Editing], page 94.

String
: A string is a kind of Lisp data object which contains a sequence of characters. Many Emacs variables are intended to have strings as values. The Lisp syntax for a string consists of the characters in the string with a '`"`' before and another '`"`' after. A '`"`' that is part of the string must be written as '`\"`' and a '`\`' that is part of the string must be written as '`\\`'. All other characters, including newline, can be included just by writing them inside the string; however, escape sequences as in C, such as '`\n`' for newline or '`\241`' using an octal character code, are allowed as well.

String Substitution
: See 'global substitution'.

Syntax Table
: The syntax table tells Emacs which characters are part of a word, which characters balance each other like parentheses, etc. See Section 28.5 [Syntax], page 229.

Tag Table
: A tag table is a file that serves as an index to the function definitions in one or more other files. See Section 21.11 [Tags], page 155.

Termscript File
: A termscript file contains a record of all characters sent by Emacs to the terminal. It is used for tracking down bugs in Emacs redisplay. Emacs does not make a termscript file unless you tell it to. See [Bugs], page 247.

Text
: Two meanings (see Chapter 20 [Text], page 121):
  - Data consisting of a sequence of characters, as opposed to binary numbers, images, graphics commands, executable programs, and the like. The contents of an Emacs buffer are always text in this sense.
  - Data consisting of written human language, as opposed to programs, or following the stylistic conventions of human language.

Top Level
: Top level is the normal state of Emacs, in which you are editing the text of the file you have visited. You are at top level whenever you are not in a recursive editing level (q.v.) or the minibuffer (q.v.), and not in the middle of a command. You can get back to top level by aborting (q.v.) and quitting (q.v.). See Section 29.1 [Quitting], page 237.

Transposition
: Transposing two units of text means putting each one into the place formerly occupied by the other. There are Emacs commands to transpose two adjacent characters, words, sexps (q.v.) or lines (see Section 14.2 [Transpose], page 83).

Truncation
: Truncating text lines in the display means leaving out any text on a line that does not fit within the right margin of the window displaying it. See also 'continuation line'. See Section 4.5 [Basic Editing], page 28.

Undoing
: Undoing means making your previous editing go in reverse, bringing back the text that existed earlier in the editing session. See Chapter 5 [Undo], page 33.

Variable
: A variable is an object in Lisp that can store an arbitrary value. Emacs uses some variables for internal purposes, and has others (known as 'options' (q.v.)) just so that you can set their values to control the behavior of Emacs. The variables used in Emacs that you are likely to be interested in are listed in the Variables Index in this manual. See Section 28.2 [Variables], page 218, for information on variables.

Visiting
: Visiting a file means loading its contents into a buffer (q.v.) where they can be edited. See Section 15.2 [Visiting], page 88.

Whitespace
: Whitespace is any run of consecutive formatting characters (space, tab, newline, and backspace).

Widening
: Widening is removing any restriction (q.v.) on the current buffer; it is the opposite of narrowing (q.v.). See Section 27.2 [Narrowing], page 208.

Window
: Emacs divides the screen into one or more windows, each of which can display the contents of one buffer (q.v.) at any time. See Chapter 1 [Screen], page 13, for basic information on how Emacs uses the screen. See Chapter 17 [Windows], page 111, for commands to control the use of windows.

Word Abbrev
: Synonymous with 'abbrev'.

Word Search
: Word search is searching for a sequence of words, considering the punctuation between them as insignificant. See Section 13.3 [Word Search], page 73.

Yanking
: Yanking means reinserting text previously killed. It can be used to undo a mistaken kill, or for copying or moving text. Some other systems call this "pasting". See Section 10.2 [Yanking], page 55.

Key (Character) Index

D

E

F

G

H

I

J

Q

R

S

T

U

W

X

Command and Function Index

S

Z

Variable Index

A

B

C

D

E

F

T

V

W

Concept Index

A

B

C

D

E

Short Contents

Table of Contents