Table of Contents

Table of Contents

Table of Contents

Table of Contents

Table of Contents

1 GNU Emacs Manual

@kbdinputstyle code

@shorttitlepage GNU Emacs Manual

GNU Emacs Manual

Thirteenth Edition, Updated for Emacs Version 20.3

Thirteenth Edition
Updated for Emacs Version 20.3,
August 1998

ISBN 1−882114−06−X

Published by the Free Software Foundation
59 Temple Place, Suite 330
Boston, MA 02111−1307 USA

Cover art by Etienne Suvasa.

2 Preface

This manual documents the use and simple customization of the Emacs editor. The reader is not expected to be a programmer; 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. For complete beginners, it is a good idea to start with the on−line, learn−by−doing tutorial, before reading the manual. To run the tutorial, start Emacs and type **`C-h t`**. This way you can 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.

On first reading, just skim chapters 1 and 2, 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 4, 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.

Chapters 14 through 19 describe intermediate−level features that are useful for all kinds of editing. Chapter 20 and following chapters describe features that you may or may not want to use; read those chapters when you need them.

Read the Trouble chapter if Emacs does not seem to be working properly. It explains how to cope with some common problems (see section Dealing with Emacs Trouble), as well as when and how to report Emacs bugs (see section Reporting Bugs). 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 is available as a printed book and also as an Info file. The Info file is for on−line perusal with the Info program, which will be the principal way of viewing documentation on−line in the GNU system. Both the Info file and the Info program itself are distributed along with GNU Emacs. The Info file and the printed book contain substantially the same text and are generated from the same source files, which are 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 GNU and Unix systems. GNU Emacs can also be used on VMS, MS−DOS (also called MS−DOG), Windows NT, and Windows 95 systems. Those systems use different file name syntax; in addition, VMS and MS−DOS do not support all GNU Emacs features. We don't try to describe VMS usage in this manual. See section Emacs and MS−DOS, for information about using Emacs on MS−DOS.

3 Distribution

GNU Emacs is *free software*; this means that everyone is free to use it and free to redistribute it on certain conditions. 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.

One way to get a copy of GNU Emacs is from someone else who has it. You need not ask for our 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 by anonymous FTP; see the file `` `etc/FTP' `` in the Emacs distribution for more information.

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.

You can also order copies of GNU Emacs from the Free Software Foundation on CD−ROM. This is a convenient and reliable way to get a copy; it is also a good way to help fund our work. (The Foundation has always received most of its funds in this way.) An order form is included in the file `` `etc/ORDERS' `` in the Emacs distribution, and on our web site in http://www.gnu.org/order/order.html. For further information, write to

```
Free Software Foundation
59 Temple Place, Suite 330
Boston, MA  02111−1307 USA
USA
```

The income from distribution fees goes to support the foundation's purpose: the development of new free software, and improvements to our existing programs including GNU Emacs.

If you find GNU Emacs useful, please ***send a donation*** to the Free Software Foundation to support our work. Donations to the Free Software Foundation are tax deductible in the US. If you use GNU Emacs at your workplace, please suggest that the company make a donation. If company policy is unsympathetic to the idea of donating to charity, you might instead suggest ordering a CD−ROM from the Foundation occasionally, or subscribing to periodic updates.

Contributors to GNU Emacs include Per Abrahamsen, Jay K. Adams, Joe Arceneaux, Boaz Ben−Zvi, Jim Blandy, Terrence Brannon, Frank Bresz, Peter Breton, Kevin Broadey, Vincent Broman, David M. Brown, Hans Chalupsky, Bob Chassell, James Clark, Mike Clarkson, Andrew Csillag, Doug Cutting, Michael DeCorte, Gary Delp, Matthieu Devin, Eri Ding, Carsten Dominik, Scott Draves, Viktor Dukhovni, John Eaton, Rolf Ebert, Stephen Eglen, Torbj@"orn Einarsson, Tsugumoto Enami, Hans Henrik Eriksen, Michael Ernst, Ata Etemadi, Frederick Farnback, Fred Fish, Karl Fogel, Gary Foster, Noah Friedman, Keith Gabryelski, Kevin Gallagher, Kevin Gallo, Howard Gayle, Stephen Gildea, David Gillespie, Boris Goldowsky, Michelangelo Grigni, Michael Gschwind, Henry Guillaume, Doug Gwyn, Ken'ichi Handa , Chris Hanson, K. Shane Hartman, John Heidemann, Markus Heritsch, Karl Heuer, Manabu Higashida, Anders Holst, Kurt Hornik, Tom Houlder, Lars Ingebrigtsen, Andrew Innes, Michael K. Johnson, Kyle Jones, Tomoji Kagatani, Brewster Kahle,

David Kaufman, Henry Kautz, Howard Kaye, Michael Kifer, Richard King, Larry K. Kolodney, Robert Krawitz, Sebastian Kremer, Geoff Kuenning, David K@aa gedal, Daniel LaLiberte, Aaron Larson, James R. Larus, Frederic Lepied, Lars Lindberg, Neil M. Mager, Ken Manheimer, Bill Mann, Brian Marick, Simon Marshall, Bengt Martensson, Charlie Martin, Thomas May, Roland McGrath, David Megginson, Wayne Mesard, Richard Mlynarik, Keith Moore, Erik Naggum, Thomas Neumann, Mike Newton, Jurgen Nickelsen, Jeff Norden, Andrew Norman, Jeff Peck, Damon Anton Permezel, Tom Perrine, Daniel Pfeiffer, Fred Pierresteguy, Christian Plaunt, Francesco A. Potorti, Michael D. Prange, Ashwin Ram, Eric S. Raymond, Paul Reilly, Edward M. Reingold, Rob Riepel, Roland B. Roberts, John Robinson, Danny Roozendaal, William Rosenblatt, Guillermo J. Rozas, Ivar Rummelhoff, Wolfgang Rupprecht, James B. Salem, Masahiko Sato, William Schelter, Ralph Schleicher, Gregor Schmid, Michael Schmidt, Ronald S. Schnell, Philippe Schnoebelen, Stephen Schoef, Randal Schwartz, Stanislav Shalunov, Mark Shapiro, Olin Shivers, Sam Shteingold, Espen Skoglund, Rick Sladkey, Lynn Slater, Chris Smith, David Smith, William Sommerfeld, Michael Staats, Ake Stenhoff, Peter Stephenson, Jonathan Stigelman, Steve Strassman, Jens T. Berger Thielemann, Spencer Thomas, Jim Thompson, Masanobu Umeda, Neil W. Van Dyke, Ulrik Vieth, Geoffrey Voelker, Johan Vromans, Barry Warsaw, Morten Welinder, Joseph Brian Wells, Ed Wilkinson, Mike Williams, Steven A. Wood, Dale R. Worley, Felix S. T. Wu, Tom Wurgler, Eli Zaretskii, Jamie Zawinski, and Neal Ziring.

4 GNU GENERAL PUBLIC LICENSE

Version 2, June 1991

4.1 Preamble

The licenses for most software are designed to take away your freedom to share and change it. By contrast, the GNU 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. This General Public License applies to most of the Free Software Foundation's software and to any other program whose authors commit to using it. (Some other Free Software Foundation software is covered by the GNU Library General Public License instead.) You can apply it to your programs, too.

When we speak of free software, we are referring to freedom, not price. Our General Public Licenses are designed to make sure that you have the freedom to distribute copies of free software (and charge for this service if you wish), 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 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 show them these terms so they know 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.

Finally, any free program is threatened constantly by software patents. We wish to avoid the danger that redistributors of a free program will individually obtain patent licenses, in effect making the program proprietary. To prevent this, we have made it clear that any patent must be licensed for everyone's free use or not licensed at all.

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

4.2 TERMS AND CONDITIONS FOR COPYING, DISTRIBUTION AND MODIFICATION

1. This License 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 derivative work under copyright law: that is to say, a work containing the Program or a portion of it, either verbatim or with modifications and/or translated into another language. (Hereinafter, translation is included without limitation in the term "modification".) Each licensee is addressed as "you". Activities other than copying, distribution and modification are not covered by this License; they are outside its scope. The act of running the Program is not restricted, and the output from the Program is covered only if its contents constitute a work based on the Program (independent of having been made by running the Program). Whether that is true depends on what the Program does.
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 License and to the absence of any warranty; and give any other recipients of the Program a copy of this License along with the Program. 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.
3. You may modify your copy or copies of the Program or any portion of it, thus forming a work based on the Program, and copy and distribute such modifications or work under the terms of Section 1 above, provided that you also meet all of these conditions:
    1. You must cause the modified files to carry prominent notices stating that you changed the files and the date of any change.
    2. You must cause any work that you distribute or publish, that in whole or in part contains or is derived from the Program or any part thereof, to be licensed as a whole at no charge to all third parties under the terms of this License.
    3. If the modified program normally reads commands interactively when run, you must cause it, when started running for such interactive use in the most ordinary 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 License. (Exception: if the Program itself is interactive but does not normally print such an announcement, your work based on the Program is not required to print an announcement.)

    These requirements apply to the modified work as a whole. If identifiable sections of that work are not derived from the Program, and can be reasonably considered independent and separate works in themselves, then this License, and its terms, do not apply to those sections when you distribute them as separate works. But when you distribute the same sections as part of a whole which is a work based on the Program, the distribution of the whole must be on the terms of this License, whose permissions for other licensees extend to the entire whole, and thus to each and every part regardless of who wrote it. Thus, it is not the intent of this section to claim rights or contest your rights to work written entirely by you; rather, the intent is to exercise the right to control the distribution of derivative or collective works based on the Program. In addition, mere aggregation of another work not based on the Program with the Program (or with a work based on the Program) on a volume of a storage or distribution medium does not bring the other work under the scope of this License.
4. You may copy and distribute the Program (or a work based on it, under Section 2) in object

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

1. Accompany it with the complete corresponding machine−readable source code, which must be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,
2. Accompany it with a written offer, valid for at least three years, to give any third party, for a charge no more than your cost of physically performing source distribution, a complete machine−readable copy of the corresponding source code, to be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,
3. Accompany it with the information you received as to the offer to distribute corresponding source code. (This alternative is allowed only for noncommercial distribution and only if you received the program in object code or executable form with such an offer, in accord with Subsection b above.)

The source code for a work means the preferred form of the work for making modifications to it. For an executable work, complete source code means all the source code for all modules it contains, plus any associated interface definition files, plus the scripts used to control compilation and installation of the executable. However, as a special exception, the source code distributed need not include anything that is normally distributed (in either source or binary form) with the major components (compiler, kernel, and so on) of the operating system on which the executable runs, unless that component itself accompanies the executable. If distribution of executable or object code is made by offering access to copy from a designated place, then offering equivalent access to copy the source code from the same place counts as distribution of the source code, even though third parties are not compelled to copy the source along with the object code.

5. You may not copy, modify, sublicense, or distribute the Program except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense or distribute the Program is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.
6. You are not required to accept this License, since you have not signed it. However, nothing else grants you permission to modify or distribute the Program or its derivative works. These actions are prohibited by law if you do not accept this License. Therefore, by modifying or distributing 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 for copying, distributing or modifying the Program or works based on it.
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. You are not responsible for enforcing compliance by third parties to this License.
8. If, as a consequence of a court judgment or allegation of patent infringement or for any other reason (not limited to patent issues), conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot distribute so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not distribute the Program at all. For example, if a patent license would not permit royalty−free redistribution of the Program by all those who receive copies directly or indirectly through you, then the only way you could satisfy both it and this License would be to refrain entirely from distribution of the Program. If any portion of this section is held invalid or unenforceable under any particular circumstance, the balance of the

section is intended to apply and the section as a whole is intended to apply in other circumstances. It is not the purpose of this section to induce you to infringe any patents or other property right claims or to contest validity of any such claims; this section has the sole purpose of protecting the integrity of the free software distribution system, which is implemented by public license practices. Many people have made generous contributions to the wide range of software distributed through that system in reliance on consistent application of that system; it is up to the author/donor to decide if he or she is willing to distribute software through any other system and a licensee cannot impose that choice. This section is intended to make thoroughly clear what is believed to be a consequence of the rest of this License.

9. If the distribution and/or use of the Program is restricted in certain countries either by patents or by copyrighted interfaces, the original copyright holder who places the Program under this License may add an explicit geographical distribution limitation excluding those countries, so that distribution is permitted only in or among countries not thus excluded. In such case, this License incorporates the limitation as if written in the body of this License.
10. 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 this 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 this License, you may choose any version ever published by the Free Software Foundation.
11. 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

12. 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.
13. 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.

4.3 END OF TERMS AND CONDITIONS

5 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 section The Organization of the Screen.

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 Basic Editing Commands.

We call Emacs advanced because it provides facilities that go beyond simple insertion and deletion: controlling subprocesses; automatic indentation of programs; viewing two or more files at once; editing formatted text; and dealing in terms of characters, words, lines, sentences, paragraphs, and pages, as well as expressions and comments in several different programming languages.

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 Help.

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 Manipulating Comments). 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 rebind the keys that way. See section Customization.

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. Almost 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. If you want to learn Emacs Lisp programming, we recommend the *Introduction to Emacs Lisp* by Robert J. Chassell, also published by the Free Software Foundation.

When run under the X Window System, Emacs provides its own menus and convenient bindings to mouse buttons. But Emacs can provide many of the benefits of a window system on a text−only terminal. For instance, you can look at or edit several files at once, move text between files, and edit files while running shell commands.

6 The Organization of the Screen

On a text−only terminal, the Emacs display occupies the whole screen. On the X Window System, Emacs creates its own X windows to use. We use the term *frame* to mean an entire text−only screen or an entire X window used by Emacs. Emacs uses both kinds of frames in the same way to display your editing. Emacs normally starts out with just one frame, but you can create additional frames if you wish. See section Frames and X Windows.

When you start Emacs, the entire frame except for the first and last lines is devoted to the text you are editing. This area is called the *window*. The first line is a *menu bar*, and the last line is a special *echo area* or *minibuffer window* where prompts appear and where you can enter responses. See below for more information about these special lines.

You can subdivide the large text window horizontally or vertically into multiple text windows, each of which can be used for a different file (see section Multiple Windows). In this manual, the word "window" always refers to the subdivisions of a frame within Emacs.

The window that the cursor is in is the *selected window*, in which editing takes place. Most Emacs commands implicitly apply to the text in the selected window (though mouse commands generally operate on whatever window you click them in, whether selected or not). The other windows display text for reference only, unless/until you select them. If you use multiple frames under the X Window System, then giving the input focus to a particular frame selects a window in that frame.

Each window's last line is a *mode line*, which describes what is going on in that window. It appears in inverse video, if the terminal supports that, and its contents begin with `` `--:-- *scratch*' `` when Emacs starts. The mode line displays status information such as what buffer is being displayed above it in the window, what major and minor modes are in use, and whether the buffer contains unsaved changes.

6.1 Point

Within Emacs, the terminal's cursor shows the location at which editing commands will take effect. This location is called *point*. Many Emacs commands move point through the text, so that you can edit at different places in it. You can also place point by clicking mouse button 1.

While the cursor appears to point *at* a character, you should think of point as *between* two characters; it points *before* the character that appears under the cursor. For example, if your text looks like `` `frob' `` with the cursor over the `` `b' ``, then point is between the `` `o' `` and the `` `b' ``. If you insert the character `` `!' `` at that position, the result is `` `fro!b' ``, with point between the `` `!' `` and the `` `b' ``. Thus, the cursor remains over the `` `b' ``, as before.

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 in its own buffer, each buffer has its own point location. A buffer that is not currently displayed remembers where point is in case you display it again later.

When there are multiple windows in a frame, 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, each window has its own position for point in that buffer.

When there are multiple frames, each frame can display one cursor. The cursor in the selected frame is solid; the cursor in other frames is a hollow box, and appears in the window that would be selected if you give the input focus to that frame.

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'.

6.2 The Echo Area

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

Echoing means displaying the characters that you type. Outside Emacs, the operating system normally echoes all your input. Emacs handles echoing differently.

Single−character commands do not echo in Emacs, and multi−character commands echo only if you pause while typing them. As soon as you pause for more than a second in the middle of a command, Emacs echoes all the characters of the command so far. This is to *prompt* you for the rest of the command. Once echoing has started, the rest of the command echoes immediately as 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 Variables Controlling Display).

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, **C−x =** prints 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.

Echo−area informative messages are saved in an editor buffer named `*Messages*'. (We have not explained buffers yet; see section Using Multiple Buffers, for more information about them.) If you miss a message that appears briefly on the screen, you can switch to the `*Messages*' buffer to see it again. (Successive progress messages are often collapsed into one in that buffer.)

The size of `*Messages*' is limited to a certain number of lines. The variable `message-log-max` specifies how many lines. Once the buffer has that many lines, each line added at the end deletes one line from the beginning. See section Variables, for how to set variables such as `message-log-max`.

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 section The Minibuffer.

6.3 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; it is the next-to-last line on the frame. The mode line is in inverse video if the terminal supports that, and it starts and ends with dashes.

Normally, the mode line looks like this:

-cs:ch buf (major minor)--line--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. For a read-only buffer, it is `%*' if the buffer is modified, and `%%' otherwise.

buf is the name of the window's *buffer*. In most cases this is the same as the name of a file you are editing. See section Using Multiple Buffers.

The buffer displayed 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.

line is `L' followed by the current line number of point. This is present when Line Number mode is enabled (which it normally is). You can optionally display the current column number too, by turning on Column Number mode (which is not enabled by default because it is somewhat slower). See section Optional Mode Line Features.

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

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, C mode, Texinfo mode, and many others. See section

Major Modes, for details of how the modes differ and how to select one.

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

minor is a list of some of the *minor modes* that are turned on at the moment in the window's chosen buffer. For example, `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 Minor Modes, 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 Narrowing. `Def' means that a keyboard macro is being defined. See section Keyboard Macros.

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 recursive editing levels affect Emacs globally, not just one buffer, the square brackets appear in every window's mode line or not in any of them. See section Recursive Editing Levels.

cs states the coding system used for the file you are editing. A dash indicates the default state of affairs: no code conversion, except for end−of−line translation if the file contents call for that. `=' means no conversion whatsoever. Nontrivial code conversions are represented by various letters−−for example, `1' refers to ISO Latin−1. See section Coding Systems, for more information. If you are using an input method, *cs* starts with a string such as `*i*>', where *i* identifies the input method. (Some input methods show `+' or `@' instead of `>'.) See section Input Methods.

When you are using a character−only terminal (not a window system), *cs* uses three characters to describe, respectively, the coding system for keyboard input, the coding system for terminal output, and the coding system used for the file you are editing.

When multibyte characters are not enabled, *cs* does not appear at all. See section Enabling Multibyte Characters.

The colon after *cs* can change to another character in certain circumstances. Emacs uses newline to separate lines in the buffer. Some files use different conventions for separating lines: either carriage−return linefeed (the MS−DOS convention) or just carriage−return (the Macintosh convention). If the buffer's file uses carriage−return linefeed, the colon changes to a backslash (`\'). If the file uses just carriage−return, the colon indicator changes to a forward slash (`/').

See section Optional Mode Line Features, for features that add other handy information to the mode line, such as the current column number of point, the current time, and whether new mail for you has arrived.

6.4 The Menu Bar

Each Emacs frame normally has a *menu bar* at the top which you can use to perform certain common operations. There's no need to list them here, as you can more easily see for yourself.

When you are using a window system, you can use the mouse to choose a command from the menu

bar. An arrow pointing right, after the menu item, indicates that the item leads to a subsidiary menu; `...' at the end means that the command will read arguments from the keyboard before it actually does anything.

To view the full command name and documentation for a menu item, type **C-h k**, and then select the menu bar with the mouse in the usual way (see section Documentation for a Key).

On text-only terminals with no mouse, you can use the menu bar by typing **M-`** or **F10** (these run the command `tmm-menubar`). This command enters a mode in which you can select a menu item from the keyboard. A provisional choice appears in the echo area. You can use the left and right arrow keys to move through the menu to different choices. When you have found the choice you want, type **RET** to select it.

Each menu item also has an assigned letter or digit which designates that item; it is usually the initial of some word in the item's name. This letter or digit is separated from the item name by `=>'. You can type the item's letter or digit to select the item.

Some of the commands in the menu bar have ordinary key bindings as well; if so, the menu lists one equivalent key binding in parentheses after the item itself.

7 Characters, Keys and Commands

This chapter explains the character sets used by Emacs for input commands and for the contents of files, and also explains the concepts of *keys* and *commands*, which are fundamental for understanding how Emacs interprets your keyboard and mouse input.

7.1 Kinds of User Input

GNU Emacs uses an extension of the ASCII character set for keyboard input; it also accepts non−character input events including function keys and mouse button actions.

ASCII consists of 128 character codes. Some of these codes are assigned graphic symbols such as `a' and `='; the rest are control characters, such as **Control−a** (usually written **C−a** for short). **C−a** gets its name from the fact that you type it by holding down the **CTRL** key while pressing **a**.

Some ASCII control characters have special names, and most terminals have special keys you can type them with: for example, **RET**, **TAB**, **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. Some keyboards have a key labeled "linefeed" which is an alias for **C−j**.

Emacs extends the ASCII character set with thousands more printing characters (see section International Character Set Support), additional control characters, and a few more modifiers that can be combined with any character.

On ASCII terminals, there are only 32 possible control characters. These are the control variants of letters and `@[]\^_'. In addition, the shift key is meaningless with control characters: **C−a** and **C−A** are the same character, and Emacs cannot distinguish them.

But the Emacs character set has room for control variants of all printing characters, and for distinguishing between **C−a** and **C−A**. X Windows makes it possible to enter all these characters. For example, **C−−** (that's Control−Minus) and **C−5** are meaningful Emacs commands under X.

Another Emacs character−set extension is additional modifier bits. Only one modifier bit is commonly used; it is called Meta. Every character has a Meta variant; examples include **Meta−a** (normally written **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 reasons of tradition, we usually write **C−M−a** rather than **M−C−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. X Windows provides several other modifier keys that can be applied to any input character. These are called **SUPER**, **HYPER** and **ALT**. We write `s−', `H−' and `A−' to say that a character uses these

modifiers. Thus, **s-H-C-x** is short for **Super-Hyper-Control-x**. Not all X terminals actually provide keys for these modifier flags--in fact, many terminals have a key labeled **ALT** which is really a **META** key. The standard key bindings of Emacs do not include any characters with these modifiers. But you can assign them meanings of your own by customizing Emacs.

Keyboard input includes keyboard keys that are not characters at all: for example function keys and arrow keys. Mouse buttons are also outside the gamut of characters. You can modify these events with the modifier keys **CTRL**, **META**, **SUPER**, **HYPER** and **ALT**, just like keyboard characters.

Input characters and non-character inputs are collectively called *input events*. See section `Input Events' in *The Emacs Lisp Reference Manual*, for more information. If you are not doing Lisp programming, but simply want to redefine the meaning of some characters or non-character events, see section Customization.

ASCII terminals cannot really send anything to the computer except ASCII characters. These terminals use a sequence of characters to represent each function key. But that is invisible to the Emacs user, because the keyboard input routines recognize these special sequences and convert them to function key events before any other part of Emacs gets to see them.

7.2 Keys

A *key sequence* (*key*, for short) is a sequence of input events that are meaningful as a unit--as "a single command." Some Emacs command sequences are just one character or one event; for example, just **C-f** is enough to move forward one character. But Emacs also has commands that take two or more events to invoke.

If a sequence of events is enough to invoke a command, it is a *complete key*. Examples of complete keys include **C-a**, **X**, **RET**, **NEXT** (a function key), **DOWN** (an arrow key), **C-x C-f**, and **C-x 4 C-f**. If it isn't long enough to be complete, we call it a *prefix key*. The above examples show that **C-x** and **C-x 4** are prefix keys. Every key sequence is either a complete key or a prefix key.

Most single characters constitute complete keys in the standard Emacs command bindings. A few of them are prefix keys. A prefix key combines with the following input event to make a longer key sequence, which may itself be complete or a prefix. For example, **C-x** is a prefix key, so **C-x** and the next input event combine to make a two-character key sequence. Most of these key sequences are complete keys, including **C-x C-f** and **C-x b**. A few, such as **C-x 4** and **C-x r**, are themselves prefix keys that lead to three-character key sequences. There's no limit to the length of a key sequence, but in practice people rarely use sequences longer than four events.

By contrast, you can't add more events onto a complete key. For example, 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. **C-f C-k** is two key sequences, not one.

All told, the prefix keys in Emacs are **C-c**, **C-h**, **C-x**, **C-x RET**, **C-x @**, **C-x a**, **C-x n**, **C-x r**, **C-x v**, **C-x 4**, **C-x 5**, **C-x 6**, **ESC**, **M-g** and **M-j**. But this list is not cast in concrete; it is just a matter of Emacs's standard key bindings. If you customize Emacs, you can make new prefix keys, or eliminate these. See section Customizing Key Bindings.

If you do make or eliminate prefix keys, that changes the set of possible key sequences. 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.

Typing the help character (**C-h** or **F1**) after a prefix character displays a list of the commands starting with that prefix. There are a few prefix characters for which **C-h** does not work--for historical reasons, they have other meanings for **C-h** which are not easy to change. But **F1** should work for all prefix characters.

7.3 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 named *commands*, and then gives keys their meanings by *binding* them to commands.

Every command has a name chosen by a programmer. The name is usually made of a few English words separated by dashes; for example, `next-line` or `forward-word`. A command also has a *function definition* which is a Lisp program; this is what makes the command do what it does. In Emacs Lisp, a command is actually a special kind of Lisp function; one which specifies how to read arguments for it and call it interactively. For more information on commands and functions, see section `What Is a Function' in *The Emacs Lisp Reference Manual*. (The definition we use in this manual is simplified slightly.)

The bindings between keys and commands are recorded in various tables called *keymaps*. See section Keymaps.

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 command `next-line` that is programmed to move down vertically. **C-n** has this effect *because* it is bound to that command. If you rebind **C-n** to the command `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 information needed for customization, 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 according to the value that you set. 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 Variables.

7.4 Character Set for Text

Text in Emacs buffers is a sequence of 8-bit bytes. Each byte can hold a single ASCII character.

Both ASCII control characters (octal codes 000 through 037, and 0177) and ASCII printing characters (codes 040 through 0176) are allowed; however, non−ASCII control characters cannot appear in a buffer. The other modifier flags used in keyboard input, such as Meta, are not allowed in buffers either.

Some ASCII control characters serve special purposes in text, and have special names. For example, the newline character (octal code 012) is used in the buffer to end a line, and the tab character (octal code 011) is used for indenting to the next tab stop column (normally every 8 columns). See section How Text Is Displayed.

Non−ASCII printing characters can also appear in buffers. When multibyte characters are enabled, you can use any of the non−ASCII printing characters that Emacs supports. They have character codes starting at 256, octal 0400, and each one is represented as a sequence of two or more bytes. See section International Character Set Support.

If you disable multibyte characters, then you can use only one alphabet of non−ASCII characters, but they all fit in one byte. They use codes 0200 through 0377. See section Single−byte European Character Support.

8 Entering and Exiting Emacs

The usual way to invoke Emacs is with the shell command `emacs'. Emacs clears the screen and then displays an initial help message and copyright notice. Some operating systems discard all type−ahead when Emacs starts up; they give Emacs no way to prevent this. Therefore, it is advisable to wait until Emacs clears the screen before typing your first editing command.

If you run Emacs from a shell window under the X Window System, run it in the background with `emacs&'. This way, Emacs does not tie up the shell window, so you can use that to run other shell commands while Emacs operates its own X windows. You can begin typing Emacs commands as soon as you direct your keyboard input to the Emacs frame.

When Emacs starts up, it makes a buffer named `*scratch*'. That's the buffer you start out in. The `*scratch*' buffer uses 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 The Init File, `~/.emacs'.)

It is 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 Command Line Arguments. But we don't recommend doing this. The feature exists mainly for compatibility with other editors.

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.

But starting a new Emacs each time you want to edit a different file does not make sense. For one thing, this would be annoyingly slow. For another, this would fail to take advantage of Emacs's ability to visit more than one file in a single editing session. And it would lose the other accumulated context, such as registers, undo history, and the mark ring.

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 session. 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. See section File Handling, for more information on visiting more than one file.

8.1 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 parent process (usually a shell), allowing you to resume editing later in the same Emacs job, with the same buffers, 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`) or iconify a frame (`iconify-or-deiconify-frame`).

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 shell command `` `%emacs' `` in most common shells.

On systems that do not support suspending programs, **C-z** starts an inferior shell that communicates directly with the terminal. Emacs waits until you exit the subshell. (The way to do that is probably with **C-d** or `` `exit' ``, but it depends on which shell you use.) 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.

Suspending also fails if you run Emacs under a shell that doesn't support suspending programs, even if the system itself does support it. In such a case, you can set the variable `cannot-suspend` to a non-`nil` value to force **C-z** to start an inferior shell. (One might also describe Emacs's parent shell as "inferior" for failing to support job control properly, but that is a matter of taste.)

When Emacs communicates directly with an X server and creates its own dedicated X windows, **C-z** has a different meaning. Suspending an applications that uses its own X windows is not meaningful or useful. Instead, **C-z** runs the command `iconify-or-deiconify-frame`, which temporarily closes up the selected Emacs frame (see section Frames and X Windows). The way to get back to a shell window is with the window manager.

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. This command first offers to save any modified file-visiting buffers. If you do not save them all, it asks for reconfirmation with **yes** before killing Emacs, since any changes not saved 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.

There is no way to restart an Emacs session once you have killed it. You can, however, arrange for Emacs to record certain session information, such as which files are visited, when you kill it, so that the next time you restart Emacs it will try to visit the same files and so on. See section Saving Emacs Sessions.

The operating system usually listens for certain special characters whose meaning is to kill or suspend the program you are running. **This operating system 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 use of **C-z** and **C-c** on several operating systems as the characters for stopping or killing a program, but that is their only relationship with the operating system. You can customize these keys to run any commands of your choice (see section Keymaps).

9 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 use the tutorial, run Emacs and type **Control−h t** (help−with−tutorial).

To clear the screen and redisplay, type **C−l** (recenter).

9.1 Inserting Text

To insert printing characters into the text you are editing, just type them. This inserts the characters you type into the buffer at the cursor (that is, at *point*; see section Point). The cursor moves forward, and any text after the cursor moves 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 deletes the preceding newline, thus joining the line with the preceding line.

Emacs can split lines automatically when they become too long, if you turn on a special minor mode called *Auto Fill* mode. See section Filling Text, for how to use Auto Fill mode.

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 Minor Modes.

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. (This character's name is normally written **C−q** for short.) There are two ways to use **C−q**:

- **C−q** followed by any non−graphic character (even **C−g**) inserts that character.
- **C−q** followed by a sequence of octal digits inserts the character with the specified octal character code. You can use any number of octal digits; any non−digit terminates the sequence. If the terminating character is **RET**, it serves only to terminate the sequence; any other non−digit is itself used as input after terminating the sequence. (The use of octal sequences is disabled in ordinary non−binary Overwrite mode, to give you a convenient way to insert a digit instead of overwriting with it.)

When multibyte characters are enabled, octal codes 0200 through 0377 are not valid as characters; if you specify a code in this range, **C−q** assumes that you intend to use some ISO Latin−*n* character set, and converts the specified code to the corresponding Emacs character code. See section Enabling Multibyte Characters. You select *which* ISO Latin character set though your choice of language environment (see section Language Environments).

To use decimal or hexadecimal instead of octal, set the variable `read-quoted-char-radix` to 10 or 16. If the radix is greater than 10, some letters starting with **a** serve as part of a character code, just like digits.

A numeric argument to **C-q** specifies how many copies of the quoted character should be inserted (see section Numeric Arguments).

Customization information: **DEL** in most modes runs the command `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.

9.2 Changing the Location of Point

To do more than insert characters, you have to know how to move point (see section Point). The simplest way to do this is with arrow keys, or by clicking the left mouse button where you want to move to.

There are also control and meta characters for cursor motion. Some are equivalent to the arrow keys (these date back to the days before terminals had arrow keys, and are usable on terminals which don't have them). Others do more sophisticated things.

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`).

M-r
: Move point to left margin, vertically centered in the window (`move-to-window-line`). Text does not move on the screen. A numeric argument says which screen line to place point on. It counts screen lines down from the top of the window (zero for the top line). A negative argument counts lines from the bottom (−1 for the bottom line).

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 Numeric Arguments, 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 point to buffer position *n*. Position 1 is the beginning of the buffer.

M-x goto-line
: Read a number *n* and move point 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 stick to a fixed 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 another line. Normally, `track-eol` is `nil`. See section Variables, for how to set variables such as `track-eol`.

Normally, **C-n** on the last line of a buffer appends a newline to it. If the variable `next-line-add-newlines` is `nil`, then **C-n** gets an error instead (like **C-p** on the first line).

9.3 Erasing Text

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

C-d
: Delete the character after point (`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 point (that is, before the cursor). Another key, **Control-d** (**C-d** for short), deletes the character after point (that is, the character that the cursor is on). This shifts the rest of the text on the line to the left. If you type **C-d** at the end of a line, it joins together that line and the next line.

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

See section Deletion and Killing, for more flexible ways of killing text.

9.4 Undoing Changes

You can undo all the recent changes in the buffer text, up to a certain point. 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.

C-u C-x u
: Undo one batch of changes in the region.

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 back to where it was before the command that made the change.

Consecutive repetitions of **C-_** or **C-x u** undo earlier and earlier changes, back to the limit of the undo information available. 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 from that moment, the previous undo commands become ordinary changes that you can undo. Thus, to redo changes you have undone, type **C-f** or any other command that will harmlessly break the sequence of undoing, then type more undo commands.

Ordinary undo applies to all changes made in the current buffer. You can also perform *selective undo*, limited to the current region. To do this, specify the region you want, then run the `undo` command with a prefix argument (the value does not matter): **C-u C-x u** or **C-u C-_**. This undoes the most recent change in the region. To undo further changes in the same region, repeat the `undo` command (no prefix argument is needed). In Transient Mark mode, any use of `undo` when there is an active region performs selective undo; you do not need a prefix argument.

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 canceled. 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.

If you do not remember whether you changed the buffer deliberately, type **C-_** once. When you see the last change you made undone, you will see whether it was an intentional change. If it was an accident, leave it undone. If it was deliberate, redo the change as described above.

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.

You cannot undo mere cursor motion; only changes in the buffer contents save undo information. However, some cursor motion commands set the mark, so if you use these commands from time to

time, you can move back to the neighborhoods you have moved through by popping the mark ring (see section The Mark Ring).

When the undo information for a buffer becomes too large, Emacs discards the oldest undo information from time to time (during garbage collection). You can specify how much undo information to keep by setting two variables: `undo-limit` and `undo-strong-limit`. Their values are expressed in units of bytes of space.

The variable `undo-limit` sets a soft limit: Emacs keeps undo data for enough commands to reach this size, and perhaps exceed it, but does not keep data for any earlier commands beyond that. Its default value is 20000. The variable `undo-strong-limit` sets a stricter limit: the command which pushes the size past this amount is itself forgotten. Its default value is 30000.

Regardless of the values of those variables, the most recent change is never discarded, so there is no danger that garbage collection occurring right after an unintentional large change might prevent you from undoing it.

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 on some keyboards it is not obvious how to type **C-_**. **C-x u** is an alternative you can type straightforwardly on any terminal.

9.5 Files

The commands described 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 section The Minibuffer).

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. If you alter the text, you can *save* the new text in the file by typing **C-x C-s** (`save-buffer`). This makes the changes permanent by copying the altered buffer contents back into the file `/usr/rms/foo.c'. Until you save, the changes exist only inside Emacs, and the file `foo.c' is unaltered.

To create a file, just visit the file with **C-x C-f** as if it already existed. This creates an empty buffer in which you can insert the text you want to put in the file. The file is actually created when you save this buffer with **C-x C-s**.

Of course, there is a lot more to learn about using files. See section File Handling.

9.6 Help

If you forget what a key does, you can find out with the Help character, which is **C-h** (or **F1**, which is an alias for **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** twice to get a description of all the help facilities. See section Help.

9.7 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 a numeric argument to tell it how many blank lines to make. See section Numeric Arguments, for how. If you have a fill prefix, then **C-o** command inserts the fill prefix on the new line, when you use it at the beginning of a line. See section The Fill Prefix.

The easy way to get rid of extra blank lines is with the command **C-x C-o** (`delete-blank-lines`). **C-x C-o** in a run of several blank lines deletes all but one of them. **C-x C-o** on a solitary blank line deletes that blank line. When point is on a nonblank line, **C-x C-o** deletes any blank lines following that nonblank line.

9.8 Continuation Lines

If you add too many characters to one line without breaking it with **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. Continuation is also called *line wrapping*.

Sometimes it is nice to have Emacs insert newlines automatically when a line gets too long. Continuation on the screen does not do that. Use Auto Fill mode (see section Filling Text) if that's what you want.

As an alternative to continuation, Emacs can display long lines 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.

Truncation instead of continuation happens whenever horizontal scrolling is in use, and optionally in all side−by−side windows (see section Multiple Windows). You can enable truncation for a particular buffer by setting the variable `truncate-lines` to non−`nil` in that buffer. (See section Variables.) 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 Local Variables.

See section Variables Controlling Display, for additional variables that affect how text is displayed.

9.9 Cursor Position Information

Here are commands to get information about the size and position of parts of the buffer, and to count lines.

`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-x line-number-mode`
: Toggle automatic display of current line number.

`M-=`
: Print number of lines in the current region (`count-lines-region`). See section The Mark and the Region, for information about the 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 working with line numbers. **`M-x what-line`** computes the current line number and displays it in the echo area. To go to a given line by number, use **`M-x goto-line`**; it prompts you for the number. These line numbers count from one at the beginning of the buffer.

You can also see the current line number in the mode line; See section The Mode Line. If you narrow the buffer, then the line number in the mode line is relative to the accessible portion (see section Narrowing). By contrast, `what-line` shows both the line number relative to the narrowed region and the line number relative to the whole buffer.

By contrast, **`M-x what-page`** counts pages from the beginning of the file, and counts lines within the page, printing both numbers. See section Pages.

While on this subject, we might as well mention **`M-=`** (`count-lines-region`), which prints the number of lines in the region (see section The Mark and the Region). See section Pages, 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: c (0143, 99, 0x63)  point=21044 of 26883(78%)  column 53
```

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

The four values after `Char:' describe the character that follows point, first by showing it and then by giving its character code in octal, decimal and hex.

`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.

`column' 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 inaccessible, **C-x =** prints additional text describing the currently accessible range. For example, it might display this:

```
Char: C (0103, 67, 0x43)  point=252 of 889(28%) 60;231 - 59962;  column 0
```

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 accessible ones. See section Narrowing.

If point is at the end of the buffer (or the end of the accessible part), **C-x =** omits any description of the character after point. The output might look like this:

```
point=26957 of 26956(100%)  column 0
```

9.10 Numeric Arguments

In mathematics and computer usage, the word *argument* means "data provided to a function or operation." You can give any Emacs command a *numeric argument* (also called a *prefix argument*). Some commands interpret the argument as a repetition count. For example, **C-f** with an argument of ten moves forward ten characters instead of one. With these commands, no argument is equivalent to an argument of one. Negative arguments tell most such commands to move or act in the opposite direction.

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 **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. Digits and **-** modified with Control, or Control and Meta, also specify numeric arguments.

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 modifier keys; **C-u** works on all terminals. To type a negative argument, type a minus sign after **C-u**. Just a minus sign without digits normally means −1.

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 Filling Text, for more information on **M-q**.) Plain **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 Deletion and Killing, 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 are described when they come up; they are always for reasons of convenience of use of the individual command.

You can use a numeric argument to insert multiple copies of a character. This is straightforward unless the character is a digit; for example, **C-u 6 4 a** inserts 64 copies of the character `a'. But this does not work for inserting digits; **C-u 6 4 1** specifies an argument of 641, rather than inserting anything. To separate the digit to insert from the argument, type another **C-u**; for example, **C-u 6 4 C-u 1** does insert 64 copies of the character `1'.

We use the term "prefix argument" as well as "numeric argument" to emphasize that you type the argument before the command, and to distinguish these arguments from minibuffer arguments that come after the command.

9.11 Repeating a Command

The command **C-x z** (`repeat`) provides another way to repeat an Emacs command many times. This command repeats the previous Emacs command, whatever that was. Repeating a command uses the same arguments that were used before; it does not read new arguments each time.

To repeat the command more than once, type additional **z**'s: each **z** repeats the command one more time. Repetition ends when you type a character other than **z**, or press a mouse button.

For example, suppose you type **C-u 2 0 C-d** to delete 20 characters. You can repeat that command (including its argument) three additional times, to delete a total of 80 characters, by typing **C-x z z z**. The first **C-x z** repeats the command once, and each subsequent **z** repeats it once again.

10 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. You can use the usual Emacs editing commands in the minibuffer to edit the argument text.

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 (for example, 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 enter a minibuffer argument is to type the text you want, terminated by **RET** which exits the minibuffer. You can cancel the command that wants the argument, and get out of the minibuffer, 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.

10.1 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 these contents:

```
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**, you will get the file named `/u2/emacs/lisp/simple.el'. Alternatively, you can kill with **M-DEL** the directory names you don't want (see section Words).

If you don't want any of the default, you can kill it with **C-a C-k**. But you don't need to kill the default; you can simply ignore it. Insert an absolute file name, one starting with a slash or a tilde, after the default directory. For example, to specify the file `/etc/termcap', just insert that name, giving these minibuffer contents:

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

Two slashes in a row are not normally meaningful in a file name, but they are allowed in GNU Emacs. They mean, "ignore everything before the second slash in the pair." Thus, `/u2/emacs/src/' is ignored in the example above, 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.

10.2 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, you can't use it to insert a newline in the minibuffer. To do that, type **C-o** or **C-q C-j**. (Recall that a newline is really the character control-J.)

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 section Multiple Windows.

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 you enable Resize-Minibuffer mode, then the minibuffer window expands vertically as necessary to hold the text that you put in the minibuffer. Use **M-x resize-minibuffer-mode** to enable or disable this minor mode (see section Minor Modes).

Scrolling works specially in the minibuffer window. When the minibuffer is just one line high, and it contains a long line of text that won't fit on the screen, scrolling automatically maintains an overlap of a certain number of characters from one continuation line to the next. The variable `minibuffer-scroll-overlap` specifies how many characters of overlap; the default is 20.

If while in the minibuffer you issue a command that displays help text of any sort in another window, you can use the **C-M-v** command while in the minibuffer to scroll the help text. This lasts until you exit the minibuffer. This feature is especially useful if a completing minibuffer gives you a list of possible completions. See section Using Other Windows.

Emacs normally disallows most commands that use the minibuffer while the minibuffer is active. This rule is to prevent recursive minibuffers from confusing novice users. If you want to be able to use such commands in the minibuffer, set the variable `enable-recursive-minibuffers` to a

non-`nil` value.

10.3 Completion

For certain kinds of arguments, you can use *completion* to enter the argument value. Completion means that you type part of the argument, then 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 rebound to complete the text 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 **M-x** uses the minibuffer to read the name of a command, it provides 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 name characters implied by the ones already present in the minibuffer, and add those characters to the ones you have given. This is what makes it possible to type **M-x ins SPC b RET** instead of **M-x insert-buffer RET** (for example).

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' does not complete to `Foo'. Completion does ignore case distinctions for certain arguments in which case does not matter.

10.3.1 Completion Example

A concrete example may help here. If you type **M-x au TAB**, the **TAB** looks for alternatives (in this case, command names) that start with `au'. There are several, including `auto-fill-mode` and `auto-save-mode`---but they are all 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 any of `cfilrs'---so no more characters are added; instead, **TAB** displays a list of all possible completions 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 fills in 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 command `minibuffer-complete` when completion is available.

10.3.2 Completion Commands

Here is a list of 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 minibuffer text, but don't go beyond 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 when completion is available runs the command `minibuffer-complete-word`.

Here are some commands you can use to choose a completion from a window that displays a list of completions:

Mouse-2
: Clicking mouse button 2 on a completion in the list of possible completions chooses that completion (`mouse-choose-completion`). You normally use this command while point is in the minibuffer; but you must click in the list of completions, not in the minibuffer itself.

PRIOR
M-v
: Typing **PRIOR** or **PAGE-UP**, or **M-v**, while in the minibuffer, selects the window showing the completion list buffer (`switch-to-completions`). This paves the way for using the commands below. (Selecting that window in the usual ways has the same effect, but this way is more convenient.)

RET
: Typing **RET** *in the completion list buffer* chooses the completion that point is in or next to (`choose-completion`). To use this command, you must first switch windows to the window that shows the list of completions.

RIGHT
: Typing the right-arrow key **RIGHT** *in the completion list buffer* moves point to the following completion (`next-completion`).

LEFT
: Typing the left-arrow key **LEFT** *in the completion list buffer* moves point toward the beginning of the buffer, to the previous completion (`previous-completion`).

10.3.3 Strict Completion

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. If the list of completions is long, you can scroll it with **C-M-v** (see section Using Other Windows).

10.3.4 Completion Options

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. However, if *all* the possible completions end in "ignored" strings, then they are not ignored. Ignored extensions do not apply to lists of completions--those always mention all possible completions.

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.

The `complete` library implements a more powerful kind of completion that can complete multiple words at a time. For example, it can complete the command name abbreviation `p-b` into `print-buffer`, because no other command starts with two words whose initials are `` `p' `` and `` `b' ``. To use this library, put `(load "complete")` in your `` `~/.emacs' `` file (see section The Init File, `` `~/.emacs' ``).

Icomplete mode presents a constantly-updated display that tells you what completions are available for the text you've entered so far. The command to enable or disable this minor mode is **M-x icomplete-mode**.

10.4 Minibuffer History

Every argument that you enter with the minibuffer is saved on a *minibuffer history list* so that you can use it again later in another argument. Special commands load the text of an earlier argument in the minibuffer. They discard the old minibuffer contents, so you can think of them as moving through the history of previous arguments.

up
M-p
: Move to the next earlier argument string saved in the minibuffer history (`previous-history-element`).

down
M-n
: Move to the next later argument string saved in the minibuffer history (`next-history-element`).

M-r *regexp* RET

Move to an earlier saved argument in the minibuffer history that has a match for *regexp* (`previous-matching-history-element`).

`M-s` *`regexp`* `RET`
Move to a later saved argument in the minibuffer history that has a match for *regexp* (`next-matching-history-element`).

The simplest way to reuse the saved arguments in the history list is to move through the history list one element at a time. While in the minibuffer, use **`M-p`** or up-arrow (`previous-history-element`) to "move to" the next earlier minibuffer input, and use **`M-n`** or down-arrow (`next-history-element`) to "move to" the next later input.

The previous input that you fetch from the history entirely replaces the contents of the minibuffer. To use it as the argument, exit the minibuffer as usual with **`RET`**. You can also edit the text before you reuse it; this does not change the history element that you "moved" to, but your new argument does go at the end of the history list in its own right.

For many minibuffer arguments there is a "default" value. In some cases, the minibuffer history commands know the default value. Then you can insert the default value into the minibuffer as text by using **`M-n`** to move "into the future" in the history. Eventually we hope to make this feature available whenever the minibuffer has a default value.

There are also commands to search forward or backward through the history; they search for history elements that match a regular expression that you specify with the minibuffer. **`M-r`** (`previous-matching-history-element`) searches older elements in the history, while **`M-s`** (`next-matching-history-element`) searches newer elements. By special dispensation, these commands can use the minibuffer to read their arguments even though you are already in the minibuffer when you issue them. As with incremental searching, an uppercase letter in the regular expression makes the search case-sensitive (see section Searching and Case).

All uses of the minibuffer record your input on a history list, but there are separate history lists for different kinds of arguments. For example, there is a list for file names, used by all the commands that read file names. (As a special feature, this history list records the absolute file name, no more and no less, even if that is not how you entered the file name.)

There are several other very specific history lists, including one for command names read by **`M-x`**, one for buffer names, one for arguments of commands like `query-replace`, and one for compilation commands read by `compile`. Finally, there is one "miscellaneous" history list that most minibuffer arguments use.

The variable `history-length` specifies the maximum length of a minibuffer history list; once a list gets that long, the oldest element is deleted each time an element is added. If the value of `history-length` is `t`, though, there is no maximum length and elements are never deleted.

10.5 Repeating Minibuffer Commands

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

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

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

C−x ESC 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 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, it will repeat exactly as before.

Once inside the minibuffer for **C−x ESC ESC**, you can use the minibuffer history commands (**M−p**, **M−n**, **M−r**, **M−s**; see section Minibuffer History) to move through the history list of saved entire commands. After finding the desired previous command, you can edit its expression as usual and then resubmit it by typing **RET** as usual.

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 re−execute a command by calling `eval` with the `command−history` element.

11 The Mark and the Region

Many Emacs commands 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*. Emacs highlights the region whenever there is one, if you enable Transient Mark mode (see section Transient Mark Mode).

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 where you put it until you set it again at another place. 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 point and the mark at opposite ends of the inserted text, 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, each buffer remembers 16 previous locations of the mark in the *mark ring*.

11.1 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`).

Drag-Mouse-1
: Set point and the mark around the text you drag across.

Mouse-3
: Set the mark where point is, then move point to where you click (`mouse-save-then-kill`).

For example, suppose you wish to convert part of the buffer to upper case, using 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**.

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.

There are two ways to set the mark with the mouse. You can drag mouse button one across a range of text; that puts point where you release the mouse button, and sets the mark at the other end of that range. Or you can click mouse button three, which sets the mark at point (like **C-SPC**) and them moves point (like **Mouse-1**). Both of these methods copy the region into the kill ring in addition to

setting the mark; that gives behavior consistent with other window-driven applications, but if you don't want to modify the kill ring, you must use keyboard commands to set the mark. See section Mouse Commands for Editing.

Ordinary terminals have only one cursor, so 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. Alternatively, 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 position of the mark. In Transient Mark mode, this command reactivates the mark.

C-x C-x is also useful when you are satisfied with the position of point but want to move the other end of the region (where the mark is); do **C-x C-x** to put point at that end of the region, and then move it. A second use of **C-x C-x**, if necessary, puts the mark at the new position with point back at its original position.

There is no such character as **C-SPC** in ASCII; when you type **SPC** while holding down **CTRL**, what you get on most ordinary terminals is the character **C-@**. This key is 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**. Under X, **C-SPC** is actually a distinct character, but its binding is still `set-mark-command`.

11.2 Transient Mark Mode

Emacs can highlight the current region, using X Windows. But normally it does not. Why not?

Highlighting the region doesn't work well ordinarily in Emacs, because once you have set a mark, there is *always* a region (in that buffer). And highlighting the region all the time would be a nuisance.

You can turn on region highlighting by enabling Transient Mark mode. This is a more rigid mode of operation in which the region "lasts" only temporarily, so you must set up a region for each command that uses one. In Transient Mark mode, most of the time there is no region; therefore, highlighting the region when it exists is convenient.

To enable Transient Mark mode, type **M-x transient-mark-mode**. This command toggles the mode, so you can repeat the command to turn off the mode.

Here are the details of Transient Mark mode:

- To set the mark, type **C-SPC** (`set-mark-command`). This makes the mark active; as you move point, you will see the region highlighting grow and shrink.
- The mouse commands for specifying the mark also make it active. So do keyboard commands whose purpose is to specify a region, including **M-@**, **C-M-@**, **M-h**, **C-M-h**, **C-x C-p**, and **C-x h**.
- When the mark is active, you can execute commands that operate on the region, such as killing, indenting, or writing to a file.
- Any change to the buffer, such as inserting or deleting a character, deactivates the mark. This means any subsequent command that operates on a region will get an error and refuse to operate. You can make the region active again by typing **C-x C-x**.

- Commands like **M->** and **C-s** that "leave the mark behind" in addition to some other primary purpose do not activate the new mark. You can activate the new region by executing **C-x C-x** (exchange-point-and-mark).
- **C-s** when the mark is active does not alter the mark.
- Quitting with **C-g** deactivates the mark.

Highlighting of the region uses the region face; you can customize how the region is highlighted by changing this face. See section Customizing Faces.

When multiple windows show the same buffer, they can have different regions, because they can have different values of point (though they all share one common mark position). Ordinarily, only the selected window highlights its region (see section Multiple Windows). However, if the variable highlight-nonselected-windows is non-nil, then each window highlights its own region (provided that Transient Mark mode is enabled and the window's buffer's mark is active).

When Transient Mark mode is not enabled, every command that sets the mark also activates it, and nothing ever deactivates it.

If the variable mark-even-if-inactive is non-nil in Transient Mark mode, then commands can use the mark and the region even when it is inactive. Region highlighting appears and disappears just as it normally does in Transient Mark mode, but the mark doesn't really go away when the highlighting disappears.

Transient Mark mode is also sometimes known as "Zmacs mode" because the Zmacs editor on the MIT Lisp Machine handled the mark in a similar way.

11.3 Operating on the Region

Once you have a region and the mark is active, here are some of the ways you can operate on the region:

- Kill it with **C-w** (see section Deletion and Killing).
- Save it in a register with **C-x r s** (see section Registers).
- Save it in a buffer or a file (see section Accumulating Text).
- Convert case with **C-x C-l** or **C-x C-u** (see section Case Conversion Commands).
- Indent it with **C-x TAB** or **C-M-** (see section Indentation).
- Fill it as text with **M-x fill-region** (see section Filling Text).
- Print hardcopy with **M-x print-region** (see section Hardcopy Output).
- Evaluate it as Lisp code with **M-x eval-region** (see section Evaluating Emacs-Lisp Expressions).

Most commands that operate on the text in the region have the word region in their names.

11.4 Commands to Mark Textual Objects

Here are the 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 commands handle arguments just like **M-f** and **C-M-f**.

Other commands set both point and mark, to delimit an object in the buffer. For example, **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 Paragraphs). It prepares the region so you can 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 Defuns). **C-x C-p** (`mark-page`) puts point before the current page, and mark at the end (see section Pages). The mark goes after the terminating page delimiter (to include it), while point goes after the preceding page delimiter (to exclude it). A numeric argument specifies a later page (if positive) or an earlier page (if negative) instead of the current page.

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.

In Transient Mark mode, all of these commands activate the mark.

11.5 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, each buffer remembers 16 previous locations of the mark, in the *mark ring*. Commands that set the mark also 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. Thus, repeated use of this command moves point to all of the old marks on the ring, one by one. The mark positions you move through in this way are not lost; they go to the end of the ring.

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 set the mark if they move point. You can tell when a command sets the mark because it displays `` `Mark Set' `` in the echo area.

If you want to move back to the same place over and over, the mark ring may not be convenient enough. If so, you can record the position in a register for later retrieval (see section Saving Positions in Registers).

The variable `mark-ring-max` specifies 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`** cycles through the positions currently in the ring.

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

11.6 The Global Mark Ring

In addition to the ordinary mark ring that belongs to each buffer, Emacs has a single *global mark ring*. It records a sequence of buffers in which you have recently set the mark, so you can go back to those buffers.

Setting the mark always makes an entry on the current buffer's mark ring. If you have switched buffers since the previous mark setting, the new mark position makes an entry on the global mark ring also. The result is that the global mark ring records a sequence of buffers that you have been in, and, for each buffer, a place where you set the mark.

The command **`C-x C-SPC`** (`pop-global-mark`) jumps to the buffer and position of the latest entry in the global ring. It also rotates the ring, so that successive uses of **`C-x C-SPC`** take you to earlier and earlier buffers.

12 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 systems use the terms "cutting" and "pasting" for these operations.

The commonest way of moving or copying text within Emacs is to kill it and later yank it elsewhere in one or more places. This is very safe because Emacs remembers several recent kills, not just the last one. It is versatile, because the many commands for killing syntactic units can also be used for moving those units. But there are other ways of copying text for special purposes.

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

12.1 Deletion and Killing

Most commands which erase text from the buffer save it in the kill ring so that you can 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 in the kill ring; they are known as *delete* commands. (This distinction is made only for erasure of text in the buffer.) If you do a kill or delete command by mistake, you can use the **C-x u** (`undo`) command to undo it (see section Undoing Changes).

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.

12.1.1 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, along with 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." This doesn't move point. **DEL** deletes the character before the cursor, and moves point back. You can delete newlines like any other characters in the buffer; deleting a newline joins two lines. Actually, **C-d** and **DEL** aren't always delete commands; when given arguments, they kill instead, since they can erase more than one character this way.

The other delete commands are those which delete only whitespace 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. If the current line is blank, it 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, by deleting a newline and all surrounding spaces, usually leaving a single space. See section Indentation.

12.1.2 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. When used on a blank line, it kills the whole line including its newline. To kill an entire non-blank line, go to the beginning and type **C-k** twice.

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 point, thus merging the next line into the current one. Spaces and tabs that you can't see 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 **C-k** will kill the newline.

When **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 *-n*, it kills *n* lines preceding the current line (together with the text on the current line before point). Thus, **C-u - 2 C-k** at the front of a line kills the two previous lines.

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

If the variable `kill-whole-line` is non-`nil`, **C-k** at the very beginning of a line kills the entire line including the following newline. This variable is normally `nil`.

12.1.3 Other Kill Commands

C-w
: Kill region (from point to the mark) (`kill-region`).

M-d
: Kill word (`kill-word`). See section Words.

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

C-x DEL
: Kill back to beginning of sentence (`backward-kill-sentence`). See section Sentences.

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

C-M-k

Kill sexp (`kill-sexp`). See section Lists and Sexps.

M-z *char*
: Kill through the 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 region around them.

A convenient way of killing is combined with searching: **M-z** (`zap-to-char`) reads a character and kills from point up to (and including) the next occurrence of that character in 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 Words); sexps, with **C-M-k** (see section Lists and Sexps); and sentences, with **C-x DEL** and **M-k** (see section Sentences).

You can use kill commands in read-only buffers. They don't actually change the buffer, and they beep to warn you of that, but they do copy the text you tried to kill into the kill ring, so you can yank it into other buffers. Most of the kill commands move point across the text they copy in this way, so that successive kill commands build up a single kill ring entry as usual.

12.2 Yanking

Yanking means reinserting text previously killed. This is what some systems call "pasting." The usual way to move or copy text is to kill it and then yank it elsewhere one or more times.

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

M-y
: Replace text just yanked with an earlier batch of killed text (`yank-pop`).

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

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

12.2.1 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, shared by 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 Accumulating Text, 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 section The Mark and the Region.

C-u C-y leaves the cursor in front of the text, and sets the mark after it. This happens only if the argument is specified with just a **C-u**, precisely. Any other sort of argument, including **C-u** and digits, specifies an earlier kill to yank (see section Yanking Earlier Kills).

To copy a block of text, you can use **M-w** (`kill-ring-save`), which copies the region into the kill ring without removing it from the buffer. This is approximately equivalent to **C-w** followed by **C-x u**, except that **M-w** does not alter the undo history and does not temporarily change the screen.

12.2.2 Appending Kills

Normally, each kill command pushes a new entry 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** yanks all the text as a unit, just as it was before it was killed.

Thus, if you want to yank text as a unit, you need not kill all of it with 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.

Commands that kill forward from point add onto the end of the previous killed text. Commands that kill backward from point add text 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 text:

```
This is a line -!-of sample text.
```

with point shown by -!-. If you type **M-d M-DEL M-d M-DEL**, killing alternately forward and backward, you end up with `` `a line of sample' `` as one entry in the kill ring, and `` `This is text.' `` in the buffer. (Note the double space, which you can clean up with **M-SPC** or **M-q**.)

Another way to kill the same text is to move back two words with **M-b M-b**, then kill all four words forward with **C-u M-d**. This produces exactly the same results in the buffer and in the kill ring. **M-f M-f C-u M-DEL** kills the same text, all going backward; once again, the result is the same. The text in the kill ring entry always has the same order that it had in the buffer before you killed it.

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`) right before 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.

A kill command following **M-w** does not append to the text that **M-w** copied into the kill ring.

12.2.3 Yanking Earlier Kills

To recover killed text that is no longer the most recent kill, use the **M-y** command (`yank-pop`). 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, first use **C-y** to yank the last kill, and then use **M-y** to replace it with the previous kill. **M-y** is allowed only after a **C-y** or another **M-y**.

You can understand **M-y** in terms of a "last yank" pointer which points at an entry in the kill ring. Each time you kill, the "last yank" pointer moves to the newly made entry at the front of the ring. **C-y** yanks the entry which the "last yank" pointer points to. **M-y** moves the "last yank" pointer to a different entry, and the text in the buffer changes to match. Enough **M-y** commands can move the

pointer to any entry in the ring, so you can get any entry into the buffer. Eventually the pointer reaches the end of the ring; the next **M-y** moves it to the first entry again.

M-y 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 entries 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 "around" to the last entry and continues 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 entry, 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 previous 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 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.

The actual contents of the kill ring are stored in a variable named `kill-ring`; you can view the entire contents of the kill ring with the command **C-h v kill-ring**.

12.3 Accumulating Text

Usually we copy or move text by killing it and yanking it, but there are other methods convenient for copying one block of text in many places, or for copying many scattered blocks of text into one place. To copy one block to many places, store it in a register (see section Registers). Here we describe the commands to accumulate scattered pieces of text into a buffer or into a file.

M-x append-to-buffer
: Append region to contents of specified 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 **M-x append-to-buffer**. This reads a buffer name, then inserts a copy of the region into the buffer specified. If you specify a nonexistent buffer, `append-to-buffer` creates the buffer. The text is inserted wherever point is in that buffer. If you have been using the buffer 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 `append-to-buffer` accumulate the text in the specified buffer in the same order as they were copied. Strictly speaking, `append-to-buffer` does not always append to the text already in the buffer--it appends only if point in that buffer is at the end. However, if `append-to-buffer` is the only command you use to alter a buffer, then point is always at the end.

`M-x prepend-to-buffer` is just like `append-to-buffer` 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.

To retrieve the accumulated text from another buffer, use the command **`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 can alternatively select the other buffer for editing, then optionally move text from it by killing. See section Using Multiple Buffers, 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 *filename* as an argument. It adds the text of the region to the end of the specified file. The file is changed immediately on disk.

You should use `append-to-file` only with files that are *not* being visited in Emacs. Using it on a file that you are editing in Emacs would change the file behind Emacs's back, which can lead to losing some of your editing.

12.4 Rectangles

The rectangle commands operate on 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, fill them with blanks or text, or delete them. Rectangle commands are useful with text in multicolumn formats, and 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 you control it in about the same way the region is controlled. But remember that a given combination of point and mark values can be interpreted either as a region or as a rectangle, depending on the command that uses them.

If point and the mark are in the same column, the rectangle they delimit is empty. If they are in the same line, the rectangle is one line high. This asymmetry between lines and columns comes about because point (and likewise the mark) is between two columns, but within a line.

`C-x r k`
: Kill the text of the region-rectangle, saving its contents as the "last killed rectangle" (`kill-rectangle`).

`C-x r d`
: Delete the text of the region-rectangle (`delete-rectangle`).

`C-x r y`

Yank the last killed rectangle with its upper left corner at point (`yank-rectangle`).

C-x r o
: Insert blank space to fill the space of the region-rectangle (`open-rectangle`). This pushes the previous contents of the region-rectangle rightward.

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

M-x delete-whitespace-rectangle
: Delete whitespace in each of the lines on the specified rectangle, starting from the left edge column of the rectangle.

C-x r t RET *string* **RET**
: Insert *string* on each line of the region-rectangle (`string-rectangle`).

The rectangle operations fall into two classes: commands deleting and inserting 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 **C-x r d** (`delete-rectangle`) and **C-x r k** (`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 into the gap.

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.

To yank the last killed rectangle, type **C-x r y** (`yank-rectangle`). Yanking a rectangle is the opposite of killing one. Point specifies where to put the rectangle's upper left corner. The rectangle's first line is inserted there, the rectangle's second line is inserted at a position one line vertically down, and so on. The number of lines affected is determined by the height of the saved rectangle.

You can convert single-column lists into double-column lists using rectangle killing and yanking; kill the second half of the list as a rectangle and then yank it beside the first line of the list. See section Two-Column Editing, for another way to edit multi-column text.

You can also copy rectangles into and out of registers with **C-x r r** *r* and **C-x r i** *r*. See section Saving Rectangles in Registers.

There are two commands you can use for making blank rectangles: **M-x clear-rectangle** which blanks out existing text, and **C-x r o** (`open-rectangle`) which inserts a blank rectangle. Clearing a rectangle is equivalent to deleting it and then inserting a blank rectangle of the same size.

The command **M-x delete-whitespace-rectangle** deletes horizontal whitespace starting from a particular column. This applies to each of the lines in the rectangle, and the column is specified by the left edge of the rectangle. The right edge of the rectangle does not make any difference to this command.

The command **C-x r t** (`M-x string-rectangle`) replaces the rectangle with a specified string (inserted once on each line). The string's width need not be the same as the width of the rectangle. if the string's width is less, the text after the rectangle shifts left; if the stringis wider than

the rectangle, the text after the rectangle shifts right.

13 Registers

Emacs *registers* are places you can save text or positions for later use. Once you save text or a rectangle in a register, you can copy it into the buffer once or many times; you can move point to a position saved in a register once or many times.

Each register has a name which is a single character. A register can store a piece of text, a rectangle, a position, a window configuration, or a file name, but only one thing at any given time. Whatever you store in a register remains there until you store something else in that register. To see what a register *r* contains, use **M-x view-register**.

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

13.1 Saving Positions in Registers

Saving a position records a place in a buffer so that you can move back there later. Moving to a saved position switches to that buffer and moves point to that place in it.

C-x r SPC *r*
: Save position of point in register *r* (`point-to-register`).

C-x r j *r*
: Jump to the position saved in register *r* (`jump-to-register`).

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

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

If you use **C-x r j** to go to a saved position, but the buffer it was saved from has been killed, **C-x r j** tries to create the buffer again by visiting the same file. Of course, this works only for buffers that were visiting files.

13.2 Saving Text in Registers

When you want to insert a copy of the same piece of text several times, it may be inconvenient to yank it from the kill ring, since each subsequent kill moves that entry further down the ring. An alternative is to store the text in a register and later retrieve it.

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

C-x r i *r*
: Insert text from register *r* (`insert-register`).

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

C-x r i *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 (**C-u**) it puts point after the text and the mark before.

13.3 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 Rectangles, for basic information on how to specify a rectangle in the buffer.

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

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

The **C-x r i *r*** command inserts a text string if the register contains one, and inserts a rectangle if the register contains one.

See also the command `sort-columns`, which you can think of as sorting a rectangle. See section Sorting Text.

13.4 Saving Window Configurations in Registers

You can save the window configuration of the selected frame in a register, or even the configuration of all windows in all frames, and restore the configuration later.

C-x r w *r*
: Save the state of the selected frame's windows in register *r* (`window-configuration-to-register`).

C-x r f *r*
: Save the state of all frames, including all their windows, in register *r* (`frame-configuration-to-register`).

Use **C-x r j *r*** to restore a window or frame configuration. This is the same command used to restore a cursor position. When you restore a frame configuration, any existing frames not included in the configuration become invisible. If you wish to delete these frames instead, use **C-u C-x r j *r***.

13.5 Keeping Numbers in Registers

There are commands to store a number in a register, to insert the number in the buffer in decimal, and to increment it. These commands can be useful in keyboard macros (see section Keyboard Macros).

C-u *number* C-x r n *reg*
: Store *number* into register *reg* (`number-to-register`).

C-u *number* C-x r + *reg*
: Increment the number in register *reg* by *number* (`increment-register`).

C-x r g *reg*
: Insert the number from register *reg* into the buffer.

C-x r g is the same command used to insert any other sort of register contents into the buffer.

13.6 Keeping File Names in Registers

If you visit certain file names frequently, you can visit them more conveniently if you put their names in registers. Here's the Lisp code used to put a file name in a register:

```
(set-register ?r '(file . name))
```

For example,

```
(set-register ?z '(file . "/gd/gnu/emacs/19.0/src/ChangeLog"))
```

puts the file name shown in register `z'.

To visit the file whose name is in register *r*, type **C-x r j *r***. (This is the same command used to jump to a position or restore a frame configuration.)

13.7 Bookmarks

Bookmarks are somewhat like registers in that they record positions you can jump to. Unlike registers, they have long names, and they persist automatically from one Emacs session to the next. The prototypical use of bookmarks is to record "where you were reading" in various files.

C-x r m RET
: Set the bookmark for the visited file, at point.

C-x r m *bookmark* RET
: Set the bookmark named *bookmark* at point (`bookmark-set`).

C-x r b *bookmark* RET
: Jump to the bookmark named *bookmark* (`bookmark-jump`).

C-x r l
: List all bookmarks (`list-bookmarks`).

M-x bookmark-save
: Save all the current bookmark values in the default bookmark file.

The prototypical use for bookmarks is to record one current position in each of several files. So the command **C-x r m**, which sets a bookmark, uses the visited file name as the default for the bookmark name. If you name each bookmark after the file it points to, then you can conveniently revisit any of those files with **C-x r b**, and move to the position of the bookmark at the same time.

To display a list of all your bookmarks in a separate buffer, type **C-x r l** (`list-bookmarks`). If you switch to that buffer, you can use it to edit your bookmark definitions or annotate the bookmarks. Type **C-h m** in that buffer for more information about its special editing commands.

When you kill Emacs, Emacs offers to save your bookmark values in your default bookmark file, `~/.emacs.bmk', if you have changed any bookmark values. You can also save the bookmarks at any time with the **M-x bookmark-save** command. The bookmark commands load your default bookmark file automatically. This saving and loading is how bookmarks persist from one Emacs session to the next.

If you set the variable `bookmark-save-flag` to 1, then each command that sets a bookmark will also save your bookmarks; this way, you don't lose any bookmark values even if Emacs crashes. (The value, if a number, says how many bookmark modifications should go by between saving.)

Bookmark position values are saved with surrounding context, so that `bookmark-jump` can find the proper position even if the file is modified slightly. The variable `bookmark-search-size` says how many characters of context to record, on each side of the bookmark's position.

Here are some additional commands for working with bookmarks:

`M-x bookmark-load RET` ***filename*** **`RET`**
: Load a file named *filename* that contains a list of bookmark values. You can use this command, as well as `bookmark-write`, to work with other files of bookmark values in addition to your default bookmark file.

`M-x bookmark-write RET` ***filename*** **`RET`**
: Save all the current bookmark values in the file *filename*.

`M-x bookmark-delete RET` ***bookmark*** **`RET`**
: Delete the bookmark named *bookmark*.

`M-x bookmark-insert-location RET` ***bookmark*** **`RET`**
: Insert in the buffer the name of the file that bookmark *bookmark* points to.

`M-x bookmark-insert RET` ***bookmark*** **`RET`**
: Insert in the buffer the *contents* of the file that bookmark *bookmark* points to.

14 Controlling the Display

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

14.1 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 portion of the text. The portion 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.

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`).

NEXT
: Likewise, scroll forward.

M−v
: Scroll backward (`scroll−down`).

PRIOR
: Likewise, scroll backward.

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

C−M−l
: Scroll heuristically to bring useful information onto the screen (`reposition−window`).

The most basic scrolling command is **C−l** (`recenter`) with no argument. It clears the entire screen and redisplays all windows. In addition, it scrolls the selected window 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. **M−v** (`scroll−down`) is like **C−v**, but moves in the opposite direction. The function keys **NEXT** and **PRIOR** are equivalent to **C−v** and **M−v**.

The names of scroll commands are based on the direction that the text moves in the window. Thus, the command to scroll forward is called `scroll−up` because it moves the text upward on the screen.

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 2.

Some users like the full-screen scroll commands to keep point at the same screen position. To enable this behavior, set the variable `scroll-preserve-screen-position` to a non-`nil` value. This mode is convenient for browsing through a file by scrolling by screenfuls; if you come back to the screen where you started, point goes back to its starting value. However, this mode is inconvenient when you move to the next screen in order to move point to the text there.

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 selected window.

The **C-M-l** command (`reposition-window`) scrolls the current window heuristically in a way designed to get useful information onto the screen. For example, in a Lisp file, this command tries to get the entire current defun onto the screen if possible.

Scrolling happens automatically if point has moved out of the visible portion of the text when it is time to display. Normally, automatic scrolling centers point vertically within the window. However, if you set `scroll-conservatively` to a small number *n*, then if you move point just a little off the screen--less than *n* lines--then Emacs scrolls the text just far enough to bring point back on screen. By default, `scroll-conservatively` is 0.

The variable `scroll-margin` restricts how close point can come to the top or bottom of a window. Its value is a number of screen lines; if point comes within that many lines of the top or bottom of the window, Emacs recenters the window. By default, `scroll-margin` is 0.

14.2 Horizontal Scrolling

Horizontal scrolling means shifting all the lines sideways within a window--so that some of the text near the left margin is not displayed at all.

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

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

When a window has been scrolled horizontally, text lines are truncated rather than continued (see section Continuation Lines), 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*. This moves part of the beginning of each line off the left edge of the window. 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 displayed normally (with each line starting at the window's left margin); attempting to do so has no effect. This means that you don't have to calculate the argument precisely for **C-x >**; any sufficiently large argument will restore the normal display.

You can request automatic horizontal scrolling by enabling Hscroll mode. When this mode is enabled, Emacs scrolls a window horizontally whenever that is necessary to keep point visible and not too far from the left or right edge. The command to enable or disable this mode is **M-x hscroll-mode**.

14.3 Follow Mode

Follow mode is a minor mode that makes two windows showing the same buffer scroll as one tall "virtual window." To use Follow mode, go to a frame with just one window, split it into two side-by-side windows using **C-x 3**, and then type **M-x follow-mode**. From then on, you can edit the buffer in either of the two windows, or scroll either one; the other window follows it.

To turn off Follow mode, type **M-x follow-mode** a second time.

14.4 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*. 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 hidden ones.

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

The hidden lines are still present in the buffer, and most editing commands see them as usual, so you may find point in the middle of the hidden 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.

To make all lines visible again, type **C-x $** with no argument.

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

14.5 Optional Mode Line Features

The current line number of point appears in the mode line when Line Number mode is enabled. Use the command **M-x line-number-mode** to turn this mode on and off; normally it is on. The line

number appears before the buffer percentage *pos*, with the letter `L' to indicate what it is. See section Minor Modes, for more information about minor modes and about how to use this command.

If the buffer is very large (larger than the value of `line-number-display-limit'), then the line number doesn't appear. Emacs doesn't compute the line number when the buffer is large, because that would be too slow. If you have narrowed the buffer (see section Narrowing), the displayed line number is relative to the accessible portion of the buffer.

You can also display the current column number by turning on Column Number mode. It displays the current column number preceded by the letter `C'. Type **M-x column-number-mode** to toggle this mode.

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 buffer name, before the mode names and their parentheses. It looks like this:

hh:*mm*pm *l.ll*

Here *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. (Some fields may be missing if your operating system cannot support them.) If you prefer time display in 24-hour format, set the variable `display-time-24hr-format' to `t'.

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

14.6 How Text Is Displayed

ASCII printing characters (octal codes 040 through 0176) in Emacs buffers are displayed with their graphics. So are non-ASCII multibyte printing characters (octal codes above 0400).

Some ASCII control characters are displayed in special ways. The newline character (octal code 012) is displayed by starting a new line. The tab character (octal code 011) is displayed by moving to the next tab stop column (normally every 8 columns).

Other ASCII control characters are normally displayed as a caret (`^') followed by the non-control version of the character; thus, control-A is displayed as `^A'.

Non-ASCII characters 0200 through 0377 are displayed with octal escape sequences; thus, character code 0243 (octal) is displayed as `\243'. However, if you enable European display, most of these characters become non-ASCII printing characters, and are displayed using their graphics (assuming your terminal supports them). See section Single-byte European Character Support.

14.7 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 section The Mode Line. If you specify the foreground color for the `modeline' face, and

`mode-line-inverse-video` is non-`nil`, then the default background color for that face is the usual foreground color. See section Using Multiple Typefaces.

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`; this tells Emacs to 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 The Echo Area.

If the variable `ctl-arrow` is `nil`, control characters in the buffer are displayed with octal escape sequences, except for 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 `Display Tables' in *The Emacs Lisp Reference Manual*.

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. The variable `tab-width` must have an integer value between 1 and 1000, inclusive.

If the variable `truncate-lines` is non-`nil`, then each line of text gets just one screen line for display; if the text line is too long, display shows only the part that fits. If `truncate-lines` is `nil`, then long text lines display as more than one screen line, enough to show the whole text of the line. See section Continuation Lines. 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`.

If the variable `truncate-partial-width-windows` is non-`nil`, it forces truncation rather than continuation in any window less than the full width of the screen or frame, regardless of the value of `truncate-lines`. For information about side-by-side windows, see section Splitting Windows. See also section `Display' in *The Emacs Lisp Reference Manual*.

The variable `baud-rate` holds the output speed of the terminal, as far as Emacs knows. Setting this variable does not change the speed of actual data transmission, but the value is used for calculations such as padding. It also affects decisions about whether to scroll part of the screen or redraw it instead--even when using a window system. (We designed it this way, despite the fact that a window system has no true "output speed," to give you a way to tune these decisions.)

You can customize the way any particular character code is displayed by means of a display table. See section `Display Tables' in *The Emacs Lisp Reference Manual*.

15 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.

15.1 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 plan to do next, you may or may not need to terminate the search explicitly with **RET**.

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. At each step, the buffer text that matches the search string is highlighted, if the terminal can do that; at each step, the current search string is updated in the echo area.

If you make a mistake in typing the search string, you can cancel 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 **RET**, 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. **RET** is necessary only if the next command you want to type is a printing character, **DEL**, **RET**, or another control character that is special within searches (**C-q**, **C-w**, **C-r**, **C-s**, **C-y**, **M-y**, **M-r**, or **M-s**).

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 aiming 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."

To reuse earlier search strings, use the *search ring*. The commands **M-p** and **M-n** move through the ring to pick a search string to reuse. These commands leave the selected search ring element in the minibuffer, where you can edit it. Type **C-s** or **C-r** to terminate editing the string and search for it.

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 **RET** 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.

An upper-case letter in the search string makes the search case-sensitive. If you delete the upper-case character from the search string, it ceases to have this effect. See section Searching and Case.

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. If you keep on going past the original starting point of the search, it changes to `Overwrapped', which means that you are revisiting matches that you have already seen.

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 newline, type **C-j**. To search for another control character, such as control-S or carriage return, you must quote it by typing **C-q** first. This function of **C-q** is analogous to its use for insertion (see section Inserting Text): it causes the following character to be treated the way any "ordinary" character is treated in the same context. You can also specify a character by its octal code: enter **C-q** followed by a sequence of octal digits.

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 canceled 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** as a key runs a command (`isearch-backward`) to search backward. A backward search finds matches that are entirely before the starting point, just as a forward search finds matches that begin after it.

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. Both **C−y** and **C−w** convert the text they copy to lower case if the search is currently not case−sensitive; this is so the search remains case−insensitive.

The character **M−y** copies text from the kill ring into the search string. It uses the same text that **C−y** as a command would yank. See section Yanking.

When you exit the incremental search, it sets the mark to where point *was*, before the search. That is convenient for moving back there. In Transient Mark mode, incremental search sets the mark without activating it, and does so only if the mark is not already active.

To customize the special characters that incremental search understands, alter their bindings in the keymap `isearch−mode−map`. For a list of bindings, look at the documentation of `isearch−mode` with **C−h f isearch−mode RET**.

15.1.1 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 you terminate the search, the single−line window is removed. Then Emacs redisplays the window in which the search was done, to show its new position of point.

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`. Its normal value is 1.

15.2 Nonincremental Search

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

C−s RET *string* RET
: Search for *string*.

C−r RET *string* RET
: Search backward for *string*.

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

The way **C−s RET** 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 RET** also works this way.

However, nonincremental searches performed using **C-s RET** 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.

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 feature that you can get to them via the incremental search commands exists for historical reasons, and to avoid the need to find suitable key sequences for them.

15.3 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 for editing a printed document made with a text formatter. 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 RET C-w *words* RET
: Search for *words*, ignoring details of punctuation.

C-r RET C-w *words* RET
: Search backward for *words*, ignoring details of punctuation.

Word search is a special case of nonincremental search and is invoked with **C-s RET 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 Regular Expression Search.

Use **C-r RET C-w** to do backward word search.

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 feature that you can get to them via the incremental search commands exists for historical reasons, and to avoid the need to find suitable key sequences for them.

15.4 Regular Expression Search

A *regular expression* (*regexp*, for short) is a pattern that denotes a class of alternative strings to match, possibly infinitely many. 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. Invoking **C-s** with a prefix argument (its value does not matter) is another way to do a forward incremental regexp search. To search backward for a regexp, use **C-M-r** (`isearch-backward-regexp`), or **C-r** with a prefix argument.

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. They also have separate search rings that you can access with **M-p** and **M-n**.

If you type **SPC** in incremental regexp search, it matches any sequence of whitespace characters, including newlines. If you want to match just a space, type **C-q SPC**.

Note that adding characters to the regexp in an incremental regexp search can make the cursor move back and start again. For example, if you have searched for `foo' and you add `\|bar', the cursor backs up in case the first `bar' precedes the first `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, or invoke them by way of incremental regexp search with **C-M-s RET** and **C-M-r RET**.

If you use the incremental regexp search commands with a prefix argument, they perform ordinary string search, like `isearch-forward` and `isearch-backward`. See section Incremental Search.

15.5 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 same character and nothing else. The special characters are `$', `^', `.', `*', `+', `?', `[', `]' and `\'. 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'. (When case distinctions are being ignored, these regexps also match `F' and `O', but we consider this a generalization of "the same string," rather than an exception.)

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 that begins with `a' and ends with `b'.

*****
: is not a construct by itself; it is a postfix operator that means to match the preceding regular expression repetitively as many times as possible. Thus, `o*' matches any number of `o's

(including no `o's). `*' always applies to the *smallest* possible preceding expression. Thus, `fo*' has a repeating `o', not a repeating `fo'. It matches `f', `fo', `foo', and so on. 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, in 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 postfix operator, similar to `*' except that it must match the preceding expression at least once. So, for example, `ca+r' matches the strings `car' and `caaaar' but not the string `cr', whereas `ca*r' matches all three strings.

?

is a postfix operator, similar to `*' except that it can match the preceding expression either once or not at all. For example, `ca?r' matches `car' or `cr'; nothing else.

[...]

is a *character set*, which begins with `[' and is terminated by `]'. In the simplest case, the characters between the two brackets are what this set can match. 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. You can also include character ranges in a character set, by writing the starting and ending characters with a `−' between them. Thus, `[a−z]' matches any lower−case ASCII letter. Ranges may be intermixed freely with individual characters, as in `[a−z$%.]', which matches any lower−case ASCII letter or `$', `%' or period. Note that the usual regexp special characters are not special 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 `−' as the first or last character of the set, or put it after a range. Thus, `[]−]' matches both `]' and `−'. To include `^' in a set, put it anywhere but at the beginning of the set. When you use a range in case−insensitive search, you should write both ends of the range in upper case, or both in lower case, or both should be non−letters. The behavior of a mixed−case range such as `A−z' is somewhat ill−defined, and it may change in future Emacs versions.

[^ ...]

`[^' begins a *complemented 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 (in other words, `−' and `]' are not special there). A complemented character set can match a newline, unless newline is mentioned as one of the characters not to match. This is in contrast to the handling of regexps in programs such as `grep'.

^

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

$

is similar to `^' but matches only at the end of a line. Thus, `x+$' 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 that matches only `$', and `\[' is a regular expression that 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; it is better to quote the special character anyway, regardless of where it appears.

For the most part, `\' followed by any character matches only that character. However, there are several exceptions: two−character sequences starting with `\' that have special meanings. The second character in the sequence is always an ordinary character when used on its own. Here is a table of `\' constructs.

\|

specifies an alternative. Two regular expressions *a* and *b* with `\|' in between form an expression that matches some text if either *a* matches it or *b* matches it. It works by trying to match *a*, and if that fails, by trying to match *b*. 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 operators `*', `+' and `?' to operate on. Thus, `ba\(na\)*' matches `bananana', etc., with any (zero or more) number of `na' strings.
3. To record 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 that is assigned as a second meaning to the same `\(... \)' construct. In practice there is no conflict between the two meanings.

d

matches the same text that matched the *d*th occurrence of a `\( ... \)' construct. 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 the digit *d* to mean "match the same text matched the *d*th time by the `\(... \)' construct." The strings matching the first nine `\( ... \)' constructs appearing in a regular expression are assigned numbers 1 through 9 in the order that the open−parentheses appear in the regular expression. So you can use `\1' through `\9' to refer to the text matched by the corresponding `\(... \)' constructs. 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. If a particular `\(... \)' construct matches more than once (which can easily happen if it is followed by `*'), only the last match is recorded.

\`

matches the empty string, but only at the beginning of the buffer or string being matched against.

\'

matches the empty string, but only at the end of the buffer or string being matched against.

\=
: matches the empty string, but only at point.

\b
: matches the empty string, but only 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 at the beginning or end of the buffer regardless of what text appears next to it.

\B
: matches the empty string, but *not* at the beginning or end of a word.

\<
: matches the empty string, but only at the beginning of a word. `\<' matches at the beginning of the buffer only if a word−constituent character follows.

\>
: matches the empty string, but only at the end of a word. `\>' matches at the end of the buffer only if the contents end with a word−constituent character.

\w
: matches any word−constituent character. The syntax table determines which characters these are. See section The Syntax Table.

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

\s*c*
: matches any character whose syntax is *c*. Here *c* is a character that represents a syntax code: thus, `w' for word constituent, `−' for whitespace, `(' for open parenthesis, etc. Represent a character of whitespace (which can be a newline) by either `−' or a space character.

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

The constructs that pertain to words and syntax are controlled by the setting of the syntax table (see section The Syntax Table).

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.

To enter the same regexp interactively, you would type **TAB** to enter a tab, and **C−j** to enter a newline. You would also type single backslashes as themselves, instead of doubling them for Lisp syntax.

15.6 Searching and Case

Incremental searches in Emacs normally ignore the case of the text they are searching through, if you specify the text in lower case. Thus, 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'.

An upper−case letter anywhere in the incremental search string makes the search case−sensitive. Thus, searching for `Foo' does not find `foo' or `FOO'. This applies to regular expression search as well as to string search. The effect ceases if you delete the upper−case letter from the search string.

If you 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 Local Variables. This variable applies to nonincremental searches also, including those performed by the replace commands (see section Replacement Commands) and the minibuffer history matching commands (see section Minibuffer History).

15.7 Replacement Commands

Global search−and−replace operations are not needed as often in Emacs as they are in other editors(1), but they are available. In addition to the simple **`M−x replace−string`** command which is like that found in most editors, there is a **`M−x query−replace`** command which asks you, for each occurrence of the pattern, whether to replace it.

The replace commands normally operate on the text from point to the end of the buffer; however, in Transient Mark mode, when the mark is active, they operate on the region. 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 Controlling Abbrev Expansion).

15.7.1 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 happens only in the text 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 Narrowing). In Transient Mark mode, when the region is active, replacement is limited to the region (see section Transient Mark Mode).

When `replace−string` exits, it leaves point at the last occurrence replaced. It sets the mark to

the prior position of point (where the `replace-string` command was issued); use **`C-u C-SPC`** to move back there.

A numeric argument restricts replacement to matches that are surrounded by word boundaries. The argument's value doesn't matter.

15.7.2 Regexp Replacement

The **`M-x replace-string`** command replaces exact matches for a single string. The similar command **`M-x 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 match being replaced. `` `\d' `` in *newstring*, where *d* is a digit, stands for whatever matched the *d*th parenthesized grouping in *regexp*. To include a `` `\' `` in the text to replace with, you must enter `` `\\' ``. For example,

```
M-x replace-regexp RET c[ad]+r RET \38;-safe RET
```

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

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

performs the inverse transformation.

15.7.3 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' ``, an all-caps `` `FOO' `` with `` `BAR' ``, and a capitalized `` `Foo' `` with `` `Bar' ``. (These three alternatives--lower case, all caps, and capitalized, are the only ones that `replace-string` can distinguish.)

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; this also inhibits case conversion of the replacement string.

15.7.4 Query Replace

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

`C-M-%` ***regexp*** **`RET`** ***newstring*** **`RET`**
`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. This preserves case, just like `replace-string', provided `case-replace' is non-`nil', as it normally is.

Aside from querying, `query-replace' works just like `replace-string', and `query-replace-regexp' works just like `replace-regexp'. This command is run by **C-M-%**.

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*.

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 to say what to do next. Since the replacement has already been made, **DEL** and **SPC** are equivalent in this situation; both move to the next occurrence. You can type **C-r** at this point (see below) to alter the replaced text. You can 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 ESC RET** to restart (see section Repeating Minibuffer Commands).

RET
: to exit without doing any more replacements.

. (Period)
: to replace this occurrence and then exit without searching for more occurrences.

!
: to replace all remaining occurrences without asking again.

^
: to go back to the position 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 meaningful, because only one previous replacement position 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** to proceed to the next occurrence. See section Recursive Editing Levels.

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** to proceed to the next occurrence.

C-l
: to redisplay the screen. Then you must type another character to specify what to do with this occurrence.

C-h
: to display a message summarizing these options. Then you must type another character to specify what to do with this occurrence.

Some other characters are aliases for the ones listed above: **y**, **n** and **q** are equivalent to **SPC**, **DEL** and **RET**.

Aside from this, any other character exits the `query-replace`, and is then reread as part of a key sequence. Thus, if you type **C-k**, it exits the `query-replace` and then kills to end of line.

To restart a `query-replace` once it is exited, use **C-x ESC ESC**, which repeats the `query-replace` because it used the minibuffer to read its arguments. See section Repeating Minibuffer Commands.

See also section Transforming File Names in Dired, for Dired commands to rename, copy, or link files by replacing regexp matches in file names.

15.8 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 RET *regexp* **RET**
: Display a list showing each line in the buffer that contains a match for *regexp*. A numeric argument specifies the number of context lines to print before and after each matching line; the default is none. To limit the search to part of the buffer, narrow to that part (see section Narrowing). If *regexp* contains upper-case letters, then the matching is done in a case-sensitive way. Otherwise, `case-fold-search` controls whether case is ignored. The buffer `` `*Occur*' `` containing the output serves as a menu for finding the occurrences in their original context. Click **Mouse-2** on an occurrence listed in `` `*Occur*' ``, or position point there and type **RET**; this switches to the buffer that was searched and moves point to the original of the chosen occurrence.

M-x list-matching-lines
: Synonym for **M-x occur**.

M-x count-matches RET *regexp* **RET**
: Print the number of matches for *regexp* after point.

M-x flush-lines RET *regexp* **RET**
: Delete each line that follows point and contains a match for *regexp*.

M-x keep-lines RET *regexp* **RET**
: Delete each line that follows point and *does not* contain a match for *regexp*.

In addition, you can use `grep` from Emacs to search a collection of files for matches for a regular expression, then visit the matches either sequentially or in arbitrary order. See section Searching with Grep under Emacs.

16 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 the fly.

The most fundamental command for correcting erroneous editing is the undo command, **C-x u** or **C-_**. This command undoes a single command (usually), a part of a command (in the case of `query-replace`), or several consecutive self-inserting characters. Consecutive repetitions of **C-_** or **C-x u** undo earlier and earlier changes, back to the limit of the undo information available. See section Undoing Changes, for for more information.

16.1 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. It deletes the character before point. When **DEL** follows a self-inserting character command, you can think of it as canceling that command. However, avoid the mistake of thinking of **DEL** as a general way to cancel a command!

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 change your mind about the phrasing of the text you are writing. **M-DEL** and **C-x DEL** save the killed text for **C-y** and **M-y** to retrieve. See section Yanking.

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. Often it requires less thought to kill the whole word and start again.

16.2 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 section Searching and Replacement.

M-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 Lists and Sexps), 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. It would change `` `f-!-oobar' `` into `` `oobf-!-ar' ``. 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.

16.3 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 Case Conversion Commands.

16.4 Checking and Correcting Spelling

This section describes the commands to check the spelling of a single word or of a portion of a buffer. These commands work with the spelling checker program Ispell, which is not part of Emacs.

M-x flyspell-mode

Enable Flyspell mode, which highlights all misspelled words.

M−$
: Check and correct spelling of the word at point (`ispell−word`).

M−TAB
: Complete the word before point based on the spelling dictionary (`ispell−complete−word`).

M−x ispell−buffer
: Check and correct spelling of each word in the buffer.

M−x ispell−region
: Check and correct spelling of each word in the region.

M−x ispell−message
: Check and correct spelling of each word in a draft mail message, excluding cited material.

M−x ispell−change−dictionary RET *dict* **RET**
: Restart the Ispell process, using *dict* as the dictionary.

M−x ispell−kill−ispell
: Kill the Ispell subprocess.

Flyspell mode is a fully−automatic way to check spelling as you edit in Emacs. It operates by checking words as you change or insert them. When it finds a word that it does not recognize, it highlights that word. This does not interfere with your editing, but when you see the highlighted word, you can move to it and fix it. Type **M−x flyspell−mode** to enable or disable this mode in the current buffer.

When Flyspell mode highlights a word as misspelled, you can click on it with **Mouse−2** to display a menu of possible corrections and actions. You can also correct the word by editing it manually in any way you like.

The other Emacs spell−checking features check or look up words when you give an explicit command to do so. Checking all or part of the buffer is useful when you have text that was written outside of this Emacs session and might contain any number of misspellings.

To check the spelling of the word around or next to point, and optionally correct it as well, use the command **M−$** (`ispell−word`). If the word is not correct, the command offers you various alternatives for what to do about it.

To check the entire current buffer, use **M−x ispell−buffer**. Use **M−x ispell−region** to check just the current region. To check spelling in an email message you are writing, use **M−x ispell−message**; that checks the whole buffer, but does not check material that is indented or appears to be cited from other messages.

Each time these commands encounter an incorrect word, they ask you what to do. They display a list of alternatives, usually including several "near−misses"−−−words that are close to the word being checked. Then you must type a character. Here are the valid responses:

SPC
: Skip this word−−continue to consider it incorrect, but don't change it here.

r *new* **RET**
: Replace the word (just this time) with *new*.

R *new* **RET**
: Replace the word with *new*, and do a `query−replace` so you can replace it elsewhere in the buffer if you wish.

digit
: Replace the word (just this time) with one of the displayed near−misses. Each near−miss is listed with a digit; type that digit to select it.

a
: Accept the incorrect word−−treat it as correct, but only in this editing session.

A
: Accept the incorrect word−−treat it as correct, but only in this editing session and for this buffer.

i
: Insert this word in your private dictionary file so that Ispell will consider it correct it from now on, even in future sessions.

u
: Insert the lower−case version of this word in your private dictionary file.

m
: Like **i**, but you can also specify dictionary completion information.

l ***word*** **RET**
: Look in the dictionary for words that match *word*. These words become the new list of "near−misses"; you can select one of them to replace with by typing a digit. You can use `*' in *word* as a wildcard.

C−g
: Quit interactive spell checking. You can restart it again afterward with **C−u M−$**.

X
: Same as **C−g**.

x
: Quit interactive spell checking and move point back to where it was when you started spell checking.

q
: Quit interactive spell checking and kill the Ispell subprocess.

C−l
: Refresh the screen.

C−z
: This key has its normal command meaning (suspend Emacs or iconify this frame).

The command `ispell−complete−word`, which is bound to the key **M−TAB** in Text mode and related modes, shows a list of completions based on spelling correction. Insert the beginning of a word, and then type **M−TAB**; the command displays a completion list window. To choose one of the completions listed, click **Mouse−2** on it, or move the cursor there in the completions window and type **RET**. See section Text Mode.

Once started, the Ispell subprocess continues to run (waiting for something to do), so that subsequent spell checking commands complete more quickly. If you want to get rid of the Ispell process, use **M−x ispell−kill−ispell**. This is not usually necessary, since the process uses no time except when you do spelling correction.

Ispell uses two dictionaries: the standard dictionary and your private dictionary. The variable `ispell−dictionary` specifies the file name of the standard dictionary to use. A value of `nil` says to use the default dictionary. The command **M−x ispell−change−dictionary** sets this variable and then restarts the Ispell subprocess, so that it will use a different dictionary.

17 File Handling

The operating system stores data permanently in named *files*. So most of the text you edit with Emacs comes from a file and is ultimately stored in a file.

To edit a file, you must tell Emacs to read 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, keep multiple versions of them, and operate on file directories.

17.1 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.) You enter the file name using the minibuffer (see section The Minibuffer). *Completion* is available, to make it easier to specify long file names. See section Completion.

For most operations, there is a *default file name* which is used if you type just **RET** to enter 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 you enter a file name without 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 file name `` `/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 you create a buffer 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. For example, if the minibuffer starts out with `/usr/tmp/' and you add `/x1/rms/foo', you get `/usr/tmp//x1/rms/foo'; but Emacs ignores everything through the first slash in the double slash; the result is `/x1/rms/foo'. See section Minibuffers for File Names.

`$' in a file name is used to substitute environment variables. For example, if you have used the shell command `export FOO=rms/hacks' to set up an environment variable named FOO, then you can use `/u/$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 shell commands to set environment variables affect Emacs only if done before Emacs is started.

To access a file with `$' in its name, type `$$'. This pair is converted to a single `$' at the same time as variable substitution is performed for single `$'. Alternatively, quote the whole file name with `/:' (see section Quoted File Names).

The Lisp function that performs the substitution is called `substitute-in-file-name`. The substitution is performed only on file names read as such using the minibuffer.

You can include non-ASCII characters in file names if you set the variable `file-name-coding-system` to a non-`nil` value. See section Specifying a Coding System.

17.2 Visiting Files

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

C-x C-r
: Visit a file for viewing, without allowing changes to it (`find-file-read-only`).

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

C-x 4 f
: Visit a file, in another window (`find-file-other-window`). Don't alter what is displayed in the selected window.

C-x 5 f
: Visit a file, in a new frame (`find-file-other-frame`). Don't alter what is displayed in the selected frame.

M-x find-file-literally
: Visit a file with no conversion of the contents.

Visiting a file means copying its contents into an Emacs buffer so you can edit them. Emacs makes a new buffer for each file that you visit. We say that this 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 editing commands 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 Saving Files.

If a buffer contains changes that have not been saved, we say the buffer is *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 to indicate that 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 section The Minibuffer), with defaulting and completion in the standard manner (see section File Names). 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 you get an error, with an error message displayed in the echo area.

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 Protection against Simultaneous Editing.

What if you want to create a new 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 the file you specify is actually a directory, **C-x C-f** invokes Dired, the Emacs directory browser, so that you can "edit" the contents of the directory (see section Dired, the Directory Editor). Dired is a convenient way to delete, look at, or operate on the files in the directory. However, if the variable `find-file-run-dired` is `nil`, then it is an error to try to visit a directory.

If you visit a file that the operating system won't let you modify, Emacs makes the buffer read-only, so that you won't go ahead and make changes that you'll have trouble saving afterward. You can make the buffer writable with **C-x C-q** (`vc-toggle-read-only`). See section Miscellaneous Buffer Operations.

Occasionally you might want to visit a file as read-only in order to protect yourself from entering changes accidentally; do so by visiting the file with the command **C-x C-r** (`find-file-read-only`).

If you visit a nonexistent file unintentionally (because you typed the wrong file name), use the **C-x C-v** command (`find-alternate-file`) to visit the file you really 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). When it reads the file name to visit, it inserts the entire default file name in the buffer, with point just after the directory part; this is convenient if you made a slight error in typing the name.

If you find a file which exists but cannot be read, **C-x C-f** signals an error.

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 buffer as before, and the other one showing the newly requested file. See section Multiple Windows.

C-x 5 f (`find-file-other-frame`) is similar, but opens a new frame, or makes visible any existing frame showing the file you seek. This feature is available only when you are using a window system. See section Frames and X Windows.

If you wish to edit a file as a sequence of characters with no special encoding or conversion, use the **M-x find-file-literally** command. It visits a file, like **C-x C-f**, but does not do format conversion (see section Editing Formatted Text), character code conversion (see section Coding Systems), or automatic uncompression (see section Accessing Compressed Files). If you already have visited the same file in the usual (non-literal) manner, this command asks you whether to visit it literally instead.

Two special hook variables 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`; this variable holds 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 a list of functions and calls them all, one by one. In both cases the functions receive no arguments. Of these two variables, `find-file-not-found-hooks` takes effect first. These variables are *not* normal hooks, and their names end in `` `-hooks' `` rather than `` `-hook' `` to indicate that fact. See section Hooks.

There are several ways to specify automatically the major mode for editing the file (see section How Major Modes are Chosen), and to specify local variables defined for that file (see section Local Variables in Files).

17.3 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 (`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** displays a message like this:

```
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** displays a message like this in the echo area:

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

The command **C-x s** (`save-some-buffers`) offers to save any or all modified buffers. It asks you what to do with each buffer. The possible responses are analogous to those of `query-replace`:

y
: Save this buffer and ask about the rest of the buffers.

n
: Don't save this buffer, but ask about the rest of the buffers.

!
: Save this buffer and all the rest with no more questions.

RET
: Terminate `save-some-buffers` without any more saving.

.
: Save this buffer, then exit `save-some-buffers` without even asking about other buffers.

C-r
: View the buffer that you are currently being asked about. When you exit View mode, you get back to `save-some-buffers`, which asks the question again.

C-h
: Display a help message about these options.

C-x C-c, the key sequence to exit Emacs, invokes `save-some-buffers` and therefore asks the same questions.

If you have changed a buffer but you do not want to save the changes, you should take some action to prevent it. Otherwise, each time you use **C-x s** or **C-x C-c**, you are liable to save this buffer 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 **M-~** 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 cancel 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 Reverting a Buffer. You could also undo all the changes by repeating the undo command **C-x u** until you have undone all the changes; but reverting is easier.

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. Then it specifies the visited file name and changes the buffer name correspondingly (as long as the new name is not in use). `set-visited-file-name` does not save the buffer in the newly visited file; it just alters the records inside Emacs in case you do save later. It also marks the buffer as "modified" so that **C-x C-s** in that buffer *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 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 the new file name implies a major mode, then **C-x C-w** switches to that major mode, in most cases. The command `set-visited-file-name` also does this. See section How Major Modes are Chosen.

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 Protection against Simultaneous Editing.

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. The default is `nil`.

17.3.1 Backup Files

On most operating systems, rewriting a file 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. (This assumes that the variable `make-backup-files` is non-`nil`. Backup files are not written if this variable is `nil`.) Emacs does not normally make backup files for files in `` `/tmp' ``.

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.

You can also explicitly request making another backup file from a buffer even though it has already been saved at least once. If you save the buffer with **C-u C-x C-s**, the version thus saved will be made into a backup file if you save the buffer again. **C-u C-u C-x C-s** saves the buffer, but first makes the previous file contents into a new backup file. **C-u C-u C-u C-x C-s** does both things: it makes a backup from the previous contents, and arranges to make another from the newly saved contents, if you save again.

17.3.1.1 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.

You can set `version-control` 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 Local Variables.

If you set the environment variable `VERSION_CONTROL`, to tell various GNU utilities what to do with backup files, Emacs also obeys the environment variable by setting the Lisp variable `version-control` accordingly at startup. If the environment variable's value is `t' or `numbered', then `version-control` becomes `t`; if the value is `nil' or `existing', then `version-control` becomes `nil`; if it is `never' or `simple', then `version-control` becomes `never`.

For files managed by a version control system (see section Version Control), the variable `vc-make-backup-files` determines whether to make backup files. By default, it is `nil`, since backup files are redundant when you store all the previous versions in a version control system. See section VC Workfile Handling.

17.3.1.2 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 `kept-old-versions` and `kept-new-versions` control this deletion. 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 `delete-old-versions` 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 Deleting Files with Dired.

17.3.1.3 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` locally (see section Local Variables in Files).

The choice of renaming or copying is controlled by three variables. Renaming is the default choice. 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 used for files that have multiple names, but renaming may still be used when the file being edited has only one name. If the variable `backup-by-copying-when-mismatch` is non-`nil`, then copying is used if renaming would cause the file's owner or group to change. `backup-by-copying-when-mismatch` is `t` by default if you start Emacs as the superuser.

When a file is managed with a version control system (see section Version Control), Emacs does not normally make backups in the usual way for that file. But check-in and check-out are similar in some ways to making backups. One unfortunate similarity is that these operations typically break hard links, disconnecting the file name you visited from any alternate names for the same file. This has nothing to do with Emacs--the version control system does it.

17.3.2 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. On all systems, Emacs checks when you save the file, and warns if you are about to overwrite another user's changes. You can prevent loss of the other user's work by taking the proper corrective action instead of saving the file.

When you make the first modification in an Emacs buffer that is visiting a file, Emacs records that the file is *locked* by you. (It does this by creating a symbolic link in the same directory with a different name.) Emacs removes the lock when you save the changes. The idea is that the file is locked whenever an Emacs buffer visiting it has unsaved changes.

If you begin to modify the buffer while the visited file is locked by someone else, this constitutes a *collision*. When Emacs detects a collision, it asks you what to do, by calling the Lisp function `ask-user-about-lock`. You can redefine this function 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 users 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, and there are cases where lock files cannot be written. In these cases, Emacs cannot detect trouble in advance, but it still can detect the collision when you try to save a file and overwrite someone else's changes.

If Emacs or the operating system crashes, this may leave behind lock files which are stale. So you may occasionally get warnings about spurious collisions. When you determine that the collision is spurious, just use **p** to tell Emacs to go ahead anyway.

Every time Emacs saves a buffer, it first checks the last-modification date of the existing file on disk to verify 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 File Directories). This shows 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.

17.4 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 reverting a buffer unintentionally could lose a lot of work, you must confirm this command with **yes**.

`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.

Reverting marks the buffer as "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 explicitly with **C-x b** cannot be reverted; `revert-buffer` reports an error when asked to do so.

When you edit a file that changes automatically and frequently--for example, a log of output from a process that continues to run--it may be useful for Emacs to revert the file without querying you, whenever you visit the file again with **C-x C-f**.

To request this behavior, set the variable `revert-without-query` to a list of regular expressions. When a file name matches one of these regular expressions, `find-file` and `revert-buffer` will revert it automatically if it has changed--provided the buffer itself is not modified. (If you have edited the text, it would be wrong to discard your changes.)

17.5 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. The message `` `Auto-saving...' `` is displayed in the echo area during auto-saving, if any files are actually auto-saved. Errors occurring during auto-saving are caught so that they do not interfere with the execution of commands you have been typing.

17.5.1 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' `` is 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.

When you delete a substantial part of the text in a large buffer, auto save turns off temporarily in that buffer. This is because if you deleted the text unintentionally, you might find the auto-save file more useful if it contains the deleted text. To reenable auto-saving after this happens, save the buffer with **C-x C-s**, or use **C-u 1 M-x auto-save**.

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.

17.5.2 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 section Entering and Exiting Emacs). 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.

Auto-saving also takes place when you stop typing for a while. The variable `auto-save-timeout` says how many seconds Emacs should wait before it does an auto save (and perhaps also a garbage collection). (The actual time period is longer if the current buffer is long; this is a heuristic which aims to keep out of your way when you are editing long buffers, in which auto-save takes an appreciable amount of time.) Auto-saving during idle periods accomplishes two things: first, it makes sure all your work is saved if you go away from the terminal for a while; second, it may avoid some auto-saving while you are actually typing.

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.

You can request an auto-save explicitly with the command **`M-x do-auto-save`**.

17.5.3 Recovering Data from Auto-Saves

You can use the contents of an auto-save file to recover from a loss of data with the command **`M-x recover-file RET`** *file* **`RET`**. This visits *file* and then (after your confirmation) restores the contents 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
yes 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.

If Emacs or the computer crashes, you can recover all the files you were editing from their auto save files with the command **`M-x recover-session`**. This first shows you a list of recorded

interrupted sessions. Move point to the one you choose, and type **C-c C-c**.

Then `recover-session` asks about each of the files that were being edited during that session, asking whether to recover that file. If you answer **y**, it calls `recover-file`, which works in its normal fashion. It shows the dates of the original file and its auto-save file, and asks once again whether to recover that file.

When `recover-session` is done, the files you've chosen to recover are present in Emacs buffers. You should then save them. Only this--saving them--updates the files themselves.

Interrupted sessions are recorded for later recovery in files named `` `~/.saves-*pid*-*hostname*' ``. The `` `~/.saves' `` portion of these names comes from the value of `auto-save-list-file-prefix`. You can arrange to record sessions in a different place by setting that variable in your `` `.emacs' `` file, but you'll have to redefine `recover-session` as well to make it look in the new place. If you set `auto-save-list-file-prefix` to `nil` in your `` `.emacs' `` file, sessions are not recorded for recovery.

17.6 File Name Aliases

Symbolic links and hard links both make it possible for several file names to refer to the same file. Hard links are alternate names that refer directly to the file; all the names are equally valid, and no one of them is preferred. By contrast, a symbolic link is a kind of defined alias: when `` `foo' `` is a symbolic link to `` `bar' ``, you can use either name to refer to the file, but `` `bar' `` is the real name, while `` `foo' `` is just an alias. More complex cases occur when symbolic links point to directories.

If you visit two names for the same file, normally Emacs makes two different buffers, but it warns you about the situation.

If you wish to avoid visiting the same file in two buffers under different names, set the variable `find-file-existing-other-name` to a non-`nil` value. Then `find-file` uses the existing buffer visiting the file, no matter which of the file's names you specify.

If the variable `find-file-visit-truename` is non-`nil`, then the file name recorded for a buffer is the file's *truename* (made by replacing all symbolic links with their target names), rather than the name you specify. Setting `find-file-visit-truename` also implies the effect of `find-file-existing-other-name`.

17.7 File Directories

The file system groups files into *directories*. A *directory listing* is a list of all the files in a directory. Emacs provides commands to create and delete directories, and to make directory listings in brief format (file names only) and verbose format (sizes, dates, and authors included). There is also a directory browser called Dired; see section Dired, the Directory Editor.

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

C-u C-x C-d *dir-or-pattern* RET
: Display a verbose directory listing.

M-x make-directory RET *dirname* RET
: Create a new directory named *dirname*.

M-x delete-directory RET *dirname* **RET**
Delete the directory named *dirname*. It must be empty, or you get an error.

The command to display 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' ``. Here is an example of specifying a file name pattern:

```
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 make a verbose listing including sizes, dates, and authors (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).

17.8 Comparing Files

The command **M-x diff** compares two files, displaying the differences in an Emacs buffer named `` `*Diff*' ``. It works by running the `diff` program, using options taken from the variable `diff-switches`, whose value should be a string.

The buffer `` `*Diff*' `` has Compilation mode as its major mode, so you can use **C-x `** to visit successive changed locations in the two source files. You can also move to a particular hunk of changes and type **RET** or **C-c C-c**, or click **Mouse-2** on it, to move to the corresponding source location. You can also use the other special commands of Compilation mode: **SPC** and **DEL** for scrolling, and **M-p** and **M-n** for cursor motion. See section Running Compilations under Emacs.

The command **M-x diff-backup** compares a specified file with its most recent backup. If you specify the name of a backup file, `diff-backup` compares it with the source file that it is a backup of.

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, and each starting position is pushed on the mark ring in its respective buffer. Then point moves forward in each window, a character at a time, until a mismatch between the two windows is reached. Then the command is finished. For more information about windows in Emacs, section Multiple Windows.

With a numeric argument, `compare-windows` ignores changes in whitespace. If the variable `compare-ignore-case` is non-`nil`, it ignores differences in case as well.

See also section Merging Files with Emerge, for convenient facilities for merging two similar files.

17.9 Miscellaneous File Operations

Emacs has commands for performing many other operations on files. All operate on one file; they do not accept wildcard 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` displays the beginning. 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 **?** while viewing for a list of them. They are mostly the same as normal Emacs cursor motion commands. To exit from viewing, type **q**. The commands for viewing are defined by a special major mode called View mode.

A related command, **M-x view-buffer**, views a buffer already present in Emacs. See section Miscellaneous Buffer Operations.

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.

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. See section Accumulating Text.

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 Dired, the Directory Editor).

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 *target* and *linkname*, then creates a symbolic link named *linkname* and pointing at *target*. The effect is that future attempts to open file *linkname* will refer to whatever file is named *target* at the time the opening is done, or will get an error if the name *target* is not in use at that time. This command does not expand the argument *target*, so that it allows you to specify a relative name as the target of the link.

Confirmation is required when creating the link if *linkname* is in use. Note that not all systems support symbolic links.

17.10 Accessing Compressed Files

Emacs comes with a library that can automatically uncompress compressed files when you visit them, and automatically recompress them if you alter them and save them. To enable this feature, type the command **`M-x auto-compression-mode`**.

When automatic compression (which implies automatic uncompression as well) is enabled, Emacs recognizes compressed files by their file names. File names ending in `` `.gz' `` indicate a file compressed with `gzip`. Other endings indicate other compression programs.

Automatic uncompression and compression apply to all the operations in which Emacs uses the contents of a file. This includes visiting it, saving it, inserting its contents into a buffer, loading it, and byte compiling it.

17.11 Remote Files

You can refer to files on other machines using a special file name syntax:

/host:filename
/user@host:filename

When you do this, Emacs uses the FTP program to read and write files on the specified host. It logs in through FTP using your user name or the name *user*. It may ask you for a password from time to time; this is used for logging in on *host*.

Normally, if you do not specify a user name in a remote file name, that means to use your own user name. But if you set the variable `ange-ftp-default-user` to a string, that string is used instead. (The Emacs package that implements FTP file access is called `ange-ftp`.)

You can entirely turn off the FTP file name feature by setting the variable `file-name-handler-alist` to `nil`.

17.12 Quoted File Names

You can *quote* an absolute file name to prevent special characters and syntax in it from having their special effects. The way to do this is to add `` `/:' `` at the beginning.

For example, you can quote a local file name which appears remote, to prevent it from being treated as a remote file name. Thus, if you have a directory named `` `/foo:' `` and a file named `` `bar' `` in it, you can refer to that file in Emacs as `` `/:/foo:/bar' ``.

`` `/:' `` can also prevent `` `~' `` from being treated as a special character for a user's home directory. For example, `` `/:/tmp/~hack' `` refers to a file whose name is `` `~hack' `` in directory `` `/tmp' ``.

Likewise, quoting with `` `/:' `` is one way to enter in the minibuffer a file name that contains `` `$' ``. However, the `` `/:' `` must be at the beginning of the buffer in order to quote `` `$' ``.

18 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 has 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 section Multiple Windows).

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 section Major Modes). 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 Local Variables.

18.1 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 *buffer* in another window (`switch-to-buffer-other-window`).

C-x 5 b *buffer* RET
: Similar, but select *buffer* in a separate frame (`switch-to-buffer-other-frame`).

To select the buffer named *bufname*, type **C-x b *bufname* RET**. This runs the command `switch-to-buffer` with argument *bufname*. You can use completion on an abbreviation for the buffer name you want (see section Completion). 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 that 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 section Major Modes).

Note that **C-x C-f**, and any other command for visiting a file, can also be used to switch to an

existing file−visiting buffer. See section Visiting Files.

Emacs uses buffer names that start with a space for internal purposes. It treats these buffers specially in minor ways−−for example, by default they do not record undo information. It is best to avoid using such buffer names yourself.

18.2 Listing Existing Buffers

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

To display 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. The buffers are listed in the order that they were current; the buffers that were current most recently come first.

`*' 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 Saving Files). `%' 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          /u2/emacs/man/
    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/'.

18.3 Miscellaneous Buffer Operations

C−x C−q
: Toggle read−only status of buffer (`vc−toggle−read−only`).

M−x rename−buffer RET ***name*** **RET**
: Change the name of the current buffer.

M−x rename−uniquely
: Rename the current buffer by adding `<*number*>' to the end.

M−x view−buffer RET ***buffer*** **RET**
: Scroll through buffer *buffer*.

A buffer can be *read−only*, which means that commands to change its contents are not allowed. The mode line indicates read−only buffers with `%%' or `%*' near the left margin. Read−only buffers are usually made by subsystems such as Dired and Rmail that have special commands to operate on the text; also by visiting a file whose access control says you cannot write it.

If you wish to make changes in a read−only buffer, use the command **C−x C−q** (`vc−toggle−read−only`). It makes a read−only buffer writable, and makes a writable

buffer read−only. In most cases, 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`. If the file is maintained with version control, **C−x C−q** works through the version control system to change the read−only status of the file as well as the buffer. See section Version Control.

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 rename−uniquely renames the current buffer to a similar name with a numeric suffix added to make it both different and unique. This command does not need an argument. It is useful for creating multiple shell buffers: if you rename the `*Shell*' buffer, then do **M−x shell** again, it makes a new shell buffer named `*Shell*'; meanwhile, the old shell buffer continues to exist under its new name. This method is also good for mail buffers, compilation buffers, and most Emacs features that create special buffers with particular names.

M−x view−buffer is much like **M−x view−file** (see section Miscellaneous File Operations) 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 with **q**, that switches back to the buffer (and the position) which was previously displayed in the window. Alternatively, if you exit View mode with **e**, the buffer and the value of point that resulted from your perusal remain in effect.

The commands **M−x append−to−buffer** and **M−x insert−buffer** can be used to copy text from one buffer to another. See section Accumulating Text.

18.4 Killing Buffers

If you continue an Emacs session for a while, you may accumulate a large number of buffers. You may then find it convenient to *kill* the buffers you no longer need. On most operating systems, killing a buffer releases its space back to the operating system so that other programs can use it. Here are some commands for killing buffers:

C−x k *bufname* RET
: Kill buffer *bufname* (`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 you kill the current buffer, another buffer is selected; one that has been selected recently but does not appear in any window now. If you ask to kill a file−visiting buffer that is modified (has unsaved editing), then you must 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`.

The buffer menu feature (see section Operating on Several Buffers) is also convenient for killing

various buffers.

If you want to do something special every time a buffer is killed, you can add hook functions to the hook `kill-buffer-hook` (see section Hooks).

If you run one Emacs session for a period of days, as many people do, it can fill up with buffers that you used several days ago. The command **`M-x clean-buffer-list`** is a convenient way to purge them; it kills all the unmodified buffers that you have not used for a long time. An ordinary buffer is killed if it has not been displayed for three days; however, you can specify certain buffers that should never be killed automatically, and others that should be killed if they have been unused for a mere hour.

You can also have this buffer purging done for you, every day at midnight, by enabling Midnight mode. Midnight mode operates each day at midnight; at that time, it runs `clean-buffer-list`, or whichever functions you have placed in the normal hook `midnight-hook` (see section Hooks).

To enable Midnight mode, use the Customization buffer to set the variable `midnight-mode` to `t`. See section Easy Customization Interface.

18.5 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 be changed only through the special commands described in this section. The usual Emacs cursor motion commands can be used in the `` `*Buffer List*' `` buffer. The following 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 you type the **`x`** command.

`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 you type the **`x`** command. You may request both saving and deletion for the same buffer.

`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.

The **d**, **C−d**, **s** and **u** commands to add or remove flags also move down (or up) one line. They accept a numeric argument as a repeat count.

These commands operate immediately on the buffer listed on the current line:

~
: Mark the buffer "unmodified." The command **~** does this immediately when you type it.

%
: Toggle the buffer's read−only flag. The command **%** does this immediately when you type it.

t
: Visit the buffer as a tags table. See section Selecting a Tags Table.

There are also commands to select another buffer or buffers:

q
: Quit the buffer menu−−immediately display the most recent formerly visible buffer in its place.

RET
f
: Immediately select this line's buffer in place of the `*Buffer List*' buffer.

o
: Immediately select this line's buffer in another window as if by **C−x 4 b**, leaving `*Buffer List*' visible.

C−o
: Immediately display this line's buffer in another window, but don't select the window.

1
: Immediately select this line's buffer in a full−screen window.

2
: Immediately set up two windows, with this line's buffer in one, and the previously selected buffer (aside from the buffer `*Buffer List*') in the other.

b
: Bury the buffer listed on this line.

m
: Mark this line's buffer to be displayed in another window if you exit with the **v** command. The request shows as a `>' at the beginning of the line. (A single buffer may not have both a delete request and a display request.)

v
: Immediately select this line's buffer, and also display in other windows any buffers previously marked with the **m** command. If you have not marked any buffers, this command is equivalent to **1**.

All that `buffer−menu` does directly is create and switch to 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 List*' 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` switches to the `*Buffer List*' buffer in the selected window; `list−buffers` displays it in another window. 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.

The buffer `*Buffer List*' is not updated automatically when buffers are created and killed; its contents are just text. If you have created, deleted or renamed buffers, the way to update `*Buffer List*' to show what you have done is to type **g** (`revert-buffer`) or repeat the `buffer-menu` command.

18.6 Indirect Buffers

An *indirect buffer* shares the text of some other buffer, which is called the *base buffer* of the indirect buffer. In some ways it is the analogue, for buffers, of a symbolic link between files.

M-x make-indirect-buffer *base-buffer* **RET** *indirect-name* **RET**
: Create an indirect buffer named *indirect-name* whose base buffer is *base-buffer*.

The text of the indirect buffer is always identical to the text of its base buffer; changes made by editing either one are visible immediately in the other. But in all other respects, the indirect buffer and its base buffer are completely separate. They have different names, different values of point, different narrowing, different markers, different major modes, and different local variables.

An indirect buffer cannot visit a file, but its base buffer can. If you try to save the indirect buffer, that actually works by saving the base buffer. Killing the base buffer effectively kills the indirect buffer, but killing an indirect buffer has no effect on its base buffer.

One way to use indirect buffers is to display multiple views of an outline. See section Viewing One Outline in Multiple Views.

19 Multiple Windows

Emacs can split a frame into two or many windows. Multiple windows can display parts of different buffers, or different parts of one buffer. Multiple frames always imply multiple windows, because each frame has its own set of windows. Each window belongs to one and only one frame.

19.1 Concepts of Emacs Windows

Each Emacs window displays one Emacs buffer at any time. A single 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. When multiple frames are visible in X Windows, each frame has a cursor which appears in the frame's selected window. The cursor in the selected frame is solid; the cursor in other frames is a hollow box.

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.

When multiple windows show the same buffer, they can have different regions, because they can have different values of point. However, they all have the same value for the mark, because each buffer has only one mark position.

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 section The Mode Line, for full details on the mode line.

@break

19.2 Splitting Windows

`C-x 2`
: Split the selected window into two windows, one above the other (`split-window-vertically`).

`C-x 3`
: Split the selected window into two windows positioned side by side (`split-window-horizontally`).

`C-Mouse-2`
: In the mode line or scroll bar of a window, split that window.

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 3 (`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. On terminals where Emacs does not support highlighting, truncated mode lines sometimes do not appear in inverse video.

You can split a window horizontally or vertically by clicking **C-Mouse-2** in the mode line or the scroll bar. The line of splitting goes through the place where you click: if you click on the mode line, the new scroll bar goes above the spot; if you click in the scroll bar, the mode line of the split window is side by side with your click.

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 Continuation Lines.

Horizontal scrolling is often used in side-by-side windows. See section Controlling the Display.

If `split-window-keep-point` is non-`nil`, the default, both of the windows resulting from **C-x 2** inherit the value of point from the window that was split. This means that scrolling is inevitable. If this variable is `nil`, then **C-x 2** tries to avoid shifting any text the screen, by putting point in each window at a position already visible in the window. It also selects whichever window contain the screen line that the cursor was previously on. Some users prefer the latter mode slow terminals.

19.3 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.

Mouse-1
: **Mouse-1**, in a window's mode line, selects that window but does not move point in it (`mouse-select-window`).

To select a different window, click with **Mouse-1** on its mode line. With the keyboard, you can switch windows by typing **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. After 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 Editing in the Minibuffer.

The usual scrolling commands (see section Controlling the Display) 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**. (In the minibuffer, **C-M-v** scrolls the window that contains the minibuffer help display, if any, rather than the next window in the standard cyclic order.)

The command `M-x compare-windows` lets you compare two files or buffers visible in two windows, by moving through them to the next mismatch. See section Comparing Files, for details.

19.4 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 C-o *bufname* **RET**
: Display buffer *bufname* in another window, but don't select that buffer or that window. This runs `display-buffer`.

C-x 4 f *filename* **RET**
: Visit file *filename* and select its buffer in another window. This runs `find-file-other-window`. See section Visiting Files.

C-x 4 d *directory* **RET**
: Select a Dired buffer for directory *directory* in another window. This runs `dired-other-window`. See section Dired, the Directory Editor.

C-x 4 m
: Start composing a mail message in another window. This runs `mail-other-window`; its same-window analogue is **C-x m** (see section Sending Mail).

C-x 4 .
: Find a tag in the current tags table, in another window. This runs `find-tag-other-window`, the multiple-window variant of **M-.** (see section Tags Tables).

C-x 4 r *filename* **RET**
: Visit file *filename* read-only, and select its buffer in another window. This runs `find-file-read-only-other-window`. See section Visiting Files.

19.5 Forcing Display in the Same Window

Certain Emacs commands switch to a specific buffer with special contents. For example, **M-x shell** switches to a buffer named `` `*Shell*' ``. By convention, all these commands are written to pop up the buffer in a separate window. But you can specify that certain of these buffers should appear in the selected window.

If you add a buffer name to the list `same-window-buffer-names`, the effect is that such commands display that particular buffer by switching to it in the selected window. For example, if

you add the element `"*grep*"` to the list, the `grep` command will display its output buffer in the selected window.

The default value of `same-window-buffer-names` is not `nil`: it specifies buffer names `` `*info*' ``, `` `*mail*' `` and `` `*shell*' `` (as well as others used by more obscure Emacs packages). This is why **`M-x shell`** normally switches to the `` `*shell*' `` buffer in the selected window. If you delete this element from the value of `same-window-buffer-names`, the behavior of **`M-x shell`** will change--it will pop up the buffer in another window instead.

You can specify these buffers more generally with the variable `same-window-regexps`. Set it to a list of regular expressions; then any buffer whose name matches one of those regular expressions is displayed by switching to it in the selected window. (Once again, this applies only to buffers that normally get displayed for you in a separate window.) The default value of this variable specifies Telnet and rlogin buffers.

An analogous feature lets you specify buffers which should be displayed in their own individual frames. See section Special Buffer Frames.

19.6 Deleting and Rearranging Windows

`C-x 0`
: Delete the selected window (`delete-window`). The last character in this key sequence is a zero.

`C-x 1`
: Delete all windows in the selected frame except the selected window (`delete-other-windows`).

`C-x 4 0`
: Delete the selected window and kill the buffer that was showing in it (`kill-buffer-and-window`). The last character in this key sequence is a zero.

`C-x ^`
: Make selected window taller (`enlarge-window`).

`C-x }`
: Make selected window wider (`enlarge-window-horizontally`).

`C-x {`
: Make selected window narrower (`shrink-window-horizontally`).

`C-x -`
: Shrink this window if its buffer doesn't need so many lines (`shrink-window-if-larger-than-buffer`).

`C-x +`
: Make all windows the same height (`balance-windows`).

`Drag-Mouse-1`
: Dragging a window's mode line up or down with **`Mouse-1`** changes window heights.

`Mouse-2`
: **`Mouse-2`** in a window's mode line deletes all other windows in the frame (`mouse-delete-other-windows`).

`Mouse-3`
: **`Mouse-3`** in a window's mode line deletes that window (`mouse-delete-window`).

To delete a window, type **`C-x 0`** (`delete-window`). (That is a zero.) The space occupied by the deleted window is given to an adjacent window (but not the minibuffer window, even if that is active

at the time). Once a window is deleted, its attributes are forgotten; only restoring a window configuration can bring it back. Deleting the window has no effect on the buffer it used to display; the buffer continues to exist, and you can select it in any window with **C-x b**.

C-x 4 0 (`kill-buffer-and-window`) is a stronger command than **C-x 0**; it kills the current buffer and then deletes the selected window.

C-x 1 (`delete-other-windows`) is more powerful in a different way; it deletes all the windows except the selected one (and the minibuffer); the selected window expands to use the whole frame except for the echo area.

You can also delete a window by clicking on its mode line with **Mouse-2**, and delete all the windows in a frame except one window by clicking on that window's mode line with **Mouse-3**.

The easiest way to adjust window heights is with a mouse. If you press **Mouse-1** on a mode line, you can drag that mode line up or down, changing the heights of the windows above and below it.

To readjust the division of space among vertically adjacent 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. **C-x {** (`shrink-window-horizontally`) makes the selected window narrower by the specified number of columns.

When you make a window bigger, the space comes from one of its neighbors. If this makes any window too small, it is deleted and its space is given to an adjacent window. The minimum size is specified by the variables `window-min-height` and `window-min-width`.

The command **C-x -** (`shrink-window-if-larger-than-buffer`) reduces the height of the selected window, if it is taller than necessary to show the whole text of the buffer it is displaying. It gives the extra lines to other windows in the frame.

You can also use **C-x +** (`balance-windows`) to even out the heights of all the windows in the selected frame.

See section Editing in the Minibuffer, for information about the Resize-Minibuffer mode, which automatically changes the size of the minibuffer window to fit the text in the minibuffer.

20 Indentation

This chapter describes the Emacs commands that add, remove, or adjust indentation.

TAB
: Indent current line "appropriately" in a mode-dependent fashion.

C-j
: Perform **RET** followed by **TAB** (`newline-and-indent`).

M-^
: Merge two lines (`delete-indentation`). This would cancel out the effect of **C-j**.

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**.

20.1 Indentation Commands and Techniques

To move over the indentation on a line, do **M-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 C-j**.

If you just want to insert a tab character in the buffer, you can type **C-q TAB**.

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 **M-^** (`delete-indentation`) command. It deletes the indentation at the front of the current line, and the line boundary as well, replacing them with a single space. As a special case (useful for Lisp code) the single space is omitted if the characters to be joined are consecutive open parentheses or closing parentheses, or if the junction follows another newline. To delete just the indentation of a line, go to the beginning of the line and use **M-** (`delete-horizontal-space`), which deletes all spaces and tabs around the cursor.

If you have a fill prefix, **M-^** deletes the fill prefix if it appears after the newline that is deleted. See section The Fill Prefix.

There are also commands for changing the indentation of several lines at once. **C-M-** (`indent-region`) applies to all the lines that begin in the region; it indents each line in the "usual" way, as if you had typed **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, `indent-relative` runs `tab-to-tab-stop` (see next section).

`indent-relative` is the definition of **TAB** in Indented Text mode. See section Commands for Human Languages.

See section Indentation in Formatted Text, for another way of specifying the indentation for part of your text.

20.2 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 command can be found on the key **M-i**.

You can specify the tab stops used by **M-i**. 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 with **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. `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 Variables Controlling Display, for more information on that.

20.3 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 Local Variables.

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.

21 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*, which means rearranging the lines of a paragraph to be approximately equal in 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. Outline mode provides special commands for operating on text with an outline structure. See section Outline Mode.

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 (see section TeX Mode). For input to nroff, use Nroff mode.

Instead of using a text formatter, you can edit formatted text in WYSIWYG style ("what you see is what you get"), with Enriched mode. Then the formatting appears on the screen in Emacs while you edit. See section Editing Formatted Text.

21.1 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 or drag a word across other words (`transpose−words`).

Notice how these keys form a series that parallels the character−based **C−f**, **C−b**, **C−d**, **DEL** and **C−t**. **M−@** is cognate to **C−@**, which is an alias for **C−SPC**.

The commands **M-f** (`forward-word`) and **M-b** (`backward-word`) move forward and backward over words. These Meta characters are thus analogous to the corresponding control characters, **C-f** and **C-b**, which move over single characters in the text. The analogy extends to numeric arguments, which serve as repeat counts. **M-f** with a negative argument moves backward, and **M-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.

M-d (`kill-word`) kills the word after point. To be precise, it kills everything from point to the place **M-f** would move to. Thus, if point is in the middle of a word, **M-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 **M-f** to get the end, and kill the word backwards with **M-DEL**.) **M-d** takes arguments just like **M-f**.

M-DEL (`backward-kill-word`) kills the word before point. It kills everything from point back to where **M-b** would move to. If point is after the space in `` `FOO, BAR' ``, then `` `FOO, ' `` is killed. (If you wish to kill just `` `FOO' ``, and not the comma and the space, use **M-b M-d** instead of **M-DEL**.)

M-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 Transposing Text, 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 **M-@** (`mark-word`) which does not move point, but sets the mark where **M-f** would move to. **M-@** accepts a numeric argument that says how many words to scan for the place to put the mark. In Transient Mark mode, this command activates the mark.

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 The Syntax Table.

21.2 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 **M-a** and **M-e** (`backward-sentence` and `forward-sentence`) move to the beginning and end of the current sentence, respectively. They were chosen to resemble **C-a** and **C-e**, which move to the beginning and end of a line. Unlike them, **M-a** and **M-e** if repeated or given numeric arguments move over successive sentences.

Moving backward over a sentence places point just before the first character of the sentence; moving forward places point right after the punctuation that ends the sentence. Neither one moves over the whitespace at the sentence boundary.

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 also a command, **C-x DEL** (`backward-kill-sentence`), for killing back to the beginning of a sentence. This command is useful when you change your mind in the middle of composing text.

The sentence commands assume that you follow the American typist's convention of putting two spaces at the end of a sentence; they consider 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.

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 Syntax of Regular Expressions.

If you want to use just one space between sentences, you should set `sentence-end` to this value:

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

You should also set the variable `sentence-end-double-space` to `nil` so that the fill commands expect and leave just one space at the end of a sentence. Note that this makes it impossible to distinguish between periods that end sentences and those that indicate abbreviations.

21.3 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`).

M-{ moves to the beginning of the current or previous paragraph, while **M-}** moves to the end of the current or next paragraph. Blank lines and text-formatter command lines separate paragraphs and are not considered part of any paragraph. In Fundamental mode, but not in Text mode, an indented line also starts a new paragraph. (If a paragraph is preceded by a blank line, these commands treat that blank line as the beginning of the paragraph.)

In major modes for programs, 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 Filling Text.

When you wish to operate on a paragraph, you can use the command **M-h** (`mark-paragraph`) to set the region around it. Thus, for example, **M-h C-w** kills the paragraph around or after point. The **M-h** command puts point at the beginning and mark at the end of the paragraph point was in. In Transient Mark mode, it activates the mark. 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.

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 (for example, blank lines). Lines that start a new paragraph and are contained in it must match only `paragraph-start`, not `paragraph-separate`. For example, in Fundamental mode, `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.

21.4 Pages

Files are often thought of as divided into *pages* by the *formfeed* character (ASCII control-L, octal code 014). When you print hardcopy for a file, this character forces a page break; thus, each page of the file goes on a separate page on paper. Most Emacs commands treat the page-separator character just like any other character: you can insert it with **C-q C-l**, and delete it with **DEL**. Thus, you are free to paginate your file or not. However, since pages are often meaningful divisions of the file, Emacs provides commands 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). **C-x C-p C-w** is a handy way to kill a page to move it elsewhere. If you move to another page delimiter with **C-x [** and **C-x]**, then yank the killed page, all the pages will be properly delimited once again. The reason **C-x C-p** includes only the following page delimiter in the region is to ensure that.

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` controls where pages begin. Its value is a regexp that matches the beginning of a line that separates pages. The normal value of this variable is `"^\f"`, which matches a formfeed character at the beginning of a line.

21.5 Filling Text

Filling text means breaking it up into lines that fit a specified width. Emacs does filling in two ways. In Auto Fill mode, inserting text with self−inserting characters also automatically fills it. There are also explicit fill commands that you can use when editing text leaves it unfilled. When you edit formatted text, you can specify a style of filling for each portion of the text (see section Editing Formatted Text).

21.5.1 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 which is enabled or disabled for each buffer individually. See section Minor Modes.

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 C-j** (recall that

a newline is really a control−J). Also, **C−o** inserts a newline without line breaking.

Auto Fill mode works well with programming−language modes, because it indents new lines 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 Manipulating Comments).

Adaptive filling (see the following section) works for Auto Filling as well as for explicit fill commands. It takes a fill prefix automatically from the second or first line of a paragraph.

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 The Init File, `~/.emacs'.

21.5.2 Explicit Fill Commands

M−q
: Fill current paragraph (`fill−paragraph`).

C−x f
: Set the fill column (`set−fill−column`).

M−x fill−region
: Fill each paragraph in the region (`fill−region`).

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 **M−q** (`fill−paragraph`). This operates on the paragraph that point is inside, or the one after point if point is between paragraphs. Refilling works by removing all the line−breaks, then inserting new ones where necessary.

To refill many paragraphs, use **M−x fill−region**, which divides the region into paragraphs and fills each of them.

M−q and `fill−region` use the same criteria as **M−h** for finding paragraph boundaries (see section Paragraphs). For more control, you can use **M−x fill−region−as−paragraph**, which refills everything between point and mark. This command deletes any blank lines within the region, so separate blocks of text end up combined into one block.

A numeric argument to **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** with no argument. (Likewise for `fill−region`.) Another way to control justification, and choose other styles of filling, is with the `justification` text property; see section Justification in Formatted Text.

The command **M−s** (`center−line`) centers the current line within the current fill column. With an argument *n*, it centers *n* 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 Local Variables. The easiest way to set `fill-column` is to use the command **C-x f** (`set-fill-column`). With a numeric argument, it uses that as the new fill column. With just **C-u** as argument, it sets `fill-column` to the current horizontal position of point.

Emacs commands normally consider a period followed by two spaces or by a newline as the end of a sentence; a period followed by just one space indicates an abbreviation and not the end of a sentence. To preserve the distinction between these two ways of using a period, the fill commands do not break a line after a period followed by just one space.

If the variable `sentence-end-double-space` is `nil`, the fill commands expect and leave just one space at the end of a sentence. Ordinarily this variable is `t`, so the fill commands insist on two spaces for the end of a sentence, as explained above. See section Sentences.

If the variable `colon-double-space` is non-`nil`, the fill commands put two spaces after a colon.

21.5.3 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), you can use the *fill prefix* feature. The fill prefix is a string that Emacs expects every line to start with, and which is not included in filling. You can specify a fill prefix explicitly; Emacs can also deduce the fill prefix automatically (see section Adaptive 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.

M-x fill-nonuniform-paragraphs
: Fill the region, considering only paragraph-separator lines 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. Auto Fill mode also inserts the fill prefix automatically when it makes a new line. The **C-o** command inserts the fill prefix on new lines it creates, when you use it at the beginning of a line (see section Blank Lines). Conversely, the command **M-^** deletes the prefix (if it occurs) after the newline that it deletes (see section Indentation).

For example, if `fill-column` is 40 and you set the fill prefix to `;; `, then **M-q** in the following text

```
;; This is an
;; example of a paragraph
;; inside a Lisp-style comment.
```

produces this:

```
;; This is an example of a paragraph
;; inside a Lisp-style comment.
```

Lines that do not start with the fill prefix are considered to start paragraphs, both in **M-q** and the paragraph commands; this gives good results for 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.

You can use **M-x fill-individual-paragraphs** to set the fill prefix for each paragraph automatically. This command divides the region into paragraphs, treating every change in the amount of indentation as the start of a new paragraph, and fills each of these paragraphs. Thus, all the lines in one "paragraph" have the same amount of indentation. That indentation serves as the fill prefix for that paragraph.

M-x fill-nonuniform-paragraphs is a similar command that divides the region into paragraphs in a different way. It considers only paragraph-separating lines (as defined by `paragraph-separate`) as starting a new paragraph. Since this means that the lines of one paragraph may have different amounts of indentation, the fill prefix used is the smallest amount of indentation of any of the lines of the paragraph. This gives good results with styles that indent a paragraph's first line more or less that the rest of the paragraph.

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 Local Variables.

The `indentation` text property provides another way to control the amount of indentation paragraphs receive. See section Indentation in Formatted Text.

21.5.4 Adaptive Filling

The fill commands can deduce the proper fill prefix for a paragraph automatically in certain cases: either whitespace or certain punctuation characters at the beginning of a line are propagated to all lines of the paragraph.

If the paragraph has two or more lines, the fill prefix is taken from the paragraph's second line, but only if it appears on the first line as well.

If a paragraph has just one line, fill commands *may* take a prefix from that line. The decision is complicated because there are three reasonable things to do in such a case:

- Use the first line's prefix on all the lines of the paragraph.
- Indent subsequent lines with whitespace, so that they line up under the text that follows the prefix on the first line, but don't actually copy the prefix from the first line.
- Don't do anything special with the second and following lines.

All three of these styles of formatting are commonly used. So the fill commands try to determine what you would like, based on the prefix that appears and on the major mode. Here is how.

If the prefix found on the first line matches `adaptive-fill-first-line-regexp`, or if it appears to be a comment-starting sequence (this depends on the major mode), then the prefix found is used for filling the paragraph, provided it would not act as a paragraph starter on subsequent lines.

Otherwise, the prefix found is converted to an equivalent number of spaces, and those spaces are used as the fill prefix for the rest of the lines, provided they would not act as a paragraph starter on subsequent lines.

In Text mode, and other modes where only blank lines and page delimiters separate paragraphs, the prefix chosen by adaptive filling never acts as a paragraph starter, so it can always be used for filling.

The variable `adaptive-fill-regexp` determines what kinds of line beginnings can serve as a fill prefix: any characters at the start of the line that match this regular expression are used. If you set the variable `adaptive-fill-mode` to `nil`, the fill prefix is never chosen automatically.

You can specify more complex ways of choosing a fill prefix automatically by setting the variable `adaptive-fill-function` to a function. This function is called with point after the left margin of a line, and it should return the appropriate fill prefix based on that line. If it returns `nil`, that means it sees no fill prefix in that line.

21.6 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. **M-l** (`downcase-word`) converts the word after point to lower case, moving past it. Thus, repeating **M-l** converts successive words. **M-u** (`upcase-word`) converts to all capitals instead, while **M-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 **M-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.

The region case conversion commands `upcase-region` and `downcase-region` are normally disabled. This means that they ask for confirmation if you try to use them. When you confirm, you may enable the command, which means it will not ask for confirmation again. See section Disabling Commands.

21.7 Text Mode

When you edit files of text in a human language, it's more convenient to use Text mode rather than Fundamental mode. To enter Text mode, type **M-x text-mode**.

In Text mode, only blank lines and page delimiters separate paragraphs. As a result, paragraphs can be indented, and adaptive filling determines what indentation to use when filling a paragraph. See section Adaptive Filling.

Text mode defines **TAB** to run `indent-relative` (see section Indentation), so that you can conveniently indent a line like the previous line. When the previous line is not indented, `indent-relative` runs `tab-to-tab-stop`, which uses Emacs tab stops that you can set (see section Tab Stops).

Text mode turns off the features concerned with comments except when you explicitly invoke them. It changes the syntax table so that periods are not considered part of a word, while apostrophes, backspaces and underlines are considered part of words.

If you indent the first lines of paragraphs, then you should use Paragraph-Indent Text mode rather than Text mode. In this mode, you do not need to have blank lines between paragraphs, because the first-line indentation is sufficient to start a paragraph; however paragraphs in which every line is indented are not supported. Use **M-x paragraph-indent-text-mode** to enter this mode.

Text mode, and all the modes based on it, define **M-TAB** as the command `ispell-complete-word`, which performs completion of the partial word in the buffer before point, using the spelling dictionary as the space of possible words. See section Checking and Correcting Spelling.

Entering Text mode runs the hook `text-mode-hook`. Other major modes related to Text mode also run this hook, followed by hooks of their own; this includes Paragraph-Indent Text mode, Nroff mode, TeX mode, Outline mode, and Mail mode. Hook functions on `text-mode-hook` can look at the value of `major-mode` to see which of these modes is actually being entered. See section Hooks.

22 Abbrevs

A defined *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'. Then you would be able to insert `find outer otter ' into the buffer by typing **f o o SPC**.

A second kind of abbreviation facility is called *dynamic abbrev expansion*. You use dynamic abbrev expansion with an explicit command to expand the letters in the buffer before point by looking for other words in the buffer that start with those letters. See section Dynamic Abbrev Expansion.

22.1 Abbrev Concepts

An *abbrev* is a word which has been defined to *expand* into a specified *expansion*. When you insert a word−separator character following the abbrev, that expands the abbrev−−replacing the abbrev with its expansion. For example, if `foo' is defined as an abbrev expanding to `find outer otter', then you can insert `find outer otter.' into the buffer by typing **f o o .**.

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 Minor Modes. `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.

22.2 Defining Abbrevs

C−x a g
: Define an abbrev, using one or more words before point as its expansion (`add−global−abbrev`).

C−x a l
: Similar, but define an abbrev specific to the current major mode (`add−mode−abbrev`).

C−x a i g
: Define a word in the buffer as an abbrev (`inverse−add−global−abbrev`).

C−x a i l
: Define a word in the buffer as a mode−specific abbrev (`inverse−add−mode−abbrev`).

M−x kill−all−abbrevs
: This command discards all abbrev definitions currently in effect, leaving a blank slate.

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 a g** (`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 a g f o o RET**.

An argument of zero to **C-x a g** means to use the contents of the region as the expansion of the abbrev being defined.

The command **C-x a l** (`add-mode-abbrev`) is similar, but defines a mode-specific abbrev. Mode-specific abbrevs are active only in a particular major mode. **C-x a l** defines an abbrev for the major mode in effect at the time **C-x a l** is typed. The arguments work the same as for **C-x a g**.

If the text already in the buffer is the abbrev, rather than its expansion, use command **C-x a i g** (`inverse-add-global-abbrev`) instead of **C-x a g**, or use **C-x a i l** (`inverse-add-mode-abbrev`) instead of **C-x a l**. These commands are called "inverse" because they invert the meaning of the two text strings they use (one from the buffer and one read with the minibuffer).

To change the definition of an abbrev, just define a new definition. When the abbrev has a prior definition, the abbrev definition commands ask for confirmation for replacing it.

To remove an abbrev definition, give a negative argument to the abbrev definition command: **C-u - C-x a g** or **C-u - C-x a l**. The former removes a global definition, while the latter removes a mode-specific definition.

M-x kill-all-abbrevs removes all the abbrev definitions there are, both global and local.

22.3 Controlling Abbrev Expansion

An abbrev expands whenever it is present in the buffer just before point and you type a self-inserting whitespace or punctuation character (**SPC**, comma, etc.). More precisely, any character that is not a word constituent expands an abbrev, and any word-constituent character can be part of an abbrev. The most common way to use an abbrev is to insert it and then insert a punctuation character to expand it.

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 commands are used to control abbrev expansion:

M-'
: Separate a prefix from a following abbrev to be expanded (`abbrev-prefix-mark`).

C-x a e
: Expand the abbrev before point (`expand-abbrev`). This is effective even when Abbrev mode is not enabled.

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 you can do is 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 hyphen 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 non-word character to expand the abbrev `cnst' into `construction'. This expansion step also deletes the hyphen that indicated **M-'** had been used. The result 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 and bring back the abbrev itself by typing **C-_** to undo (see section Undoing Changes). This also undoes the insertion of the non-word character that expanded the abbrev. If the result you want is the terminating non-word character plus the unexpanded abbrev, you must reinsert the terminating character, quoting it with **C-q**. You can also use the command **M-x unexpand-abbrev** to cancel the last expansion without deleting the terminating 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.

Expanding an abbrev runs the hook pre-abbrev-expand-hook (see section Hooks).

22.4 Examining and Editing Abbrevs

M-x list-abbrevs
: Display 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 of the line is the abbrev. The number that follows is the number of times the abbrev has been expanded. Emacs keeps track of this to help you see which abbrevs you actually use, so that you can 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. Type **C-c C-c** in this buffer to install the abbrev definitions as specified in the buffer--and delete any abbrev definitions not listed.

The command `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.

22.5 Saving Abbrevs

These commands allow you to keep abbrev definitions between editing sessions.

M-x write-abbrev-file RET *file* **RET**
: Write a file *file* describing all defined abbrevs.

M-x read-abbrev-file RET *file* **RET**
: Read the file *file* and define abbrevs as specified therein.

M-x quietly-read-abbrev-file RET *file* **RET**
: Similar but do not display a message about what is going on.

M-x define-abbrevs
: Define abbrevs from definitions in current buffer.

M-x insert-abbrevs
: Insert all abbrevs and their expansions into current buffer.

M-x write-abbrev-file reads a file name using the minibuffer and then writes a description of all current abbrev definitions into that file. This is used to save abbrev definitions for use in a later session. The text stored in the file is a series of Lisp expressions that, when executed, define the same abbrevs that you currently have.

M-x read-abbrev-file reads a file name using the minibuffer and then 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, they use the file name specified in 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.

22.6 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`).

C-M-/
: Complete the word before point as a dynamic abbrev (`dabbrev-completion`).

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 scanning all the text before point, it searches the text after point. The variable `dabbrev-limit`, if non-`nil`, specifies how far in the buffer to search for an expansion.

After scanning the current buffer, **M-/** normally searches other buffers, unless you have set `dabbrev-check-all-buffers` to `nil`.

A negative argument to **M-/**, as in **C-u - M-/**, says to search first for expansions after point, and second for expansions before point. If you repeat the **M-/** to look for another expansion, do not specify an argument. This tries all the expansions after point and then the expansions before point.

After you have expanded a dynamic abbrev, you can copy additional words that follow the expansion in its original context. Simply type **SPC M-/** for each word you want to copy. The spacing and punctuation between words is copied along with the words.

The command **C-M-/** (`dabbrev-completion`) performs completion of a dynamic abbreviation. Instead of trying the possible expansions one by one, it finds all of them, then inserts the text that they have in common. If they have nothing in common, **C-M-/** displays a list of completions, from which you can select a choice in the usual manner. See section Completion.

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.

22.7 Customizing Dynamic Abbreviation

Normally, dynamic abbrev expansion ignores case when searching for expansions. That is, the expansion need not agree in case with the word you are expanding.

This feature is controlled by the variable `dabbrev-case-fold-search`. If it is `t`, case is ignored in this search; if `nil`, the word and the expansion must match in case. If the value of `dabbrev-case-fold-search` is `case-fold-search`, which is true by default, then the variable `case-fold-search` controls whether to ignore case while searching for expansions.

Normally, dynamic abbrev expansion preserves the case pattern *of the abbrev you have typed*, by converting the expansion to that case pattern.

The variable `dabbrev-case-replace` controls whether to preserve the case pattern of the abbrev. If it is `t`, the abbrev's case pattern is preserved in most cases; if `nil`, the expansion is always copied verbatim. If the value of `dabbrev-case-replace` is `case-replace`, which is true by default, then the variable `case-replace` controls whether to copy the expansion verbatim.

However, if the expansion contains a complex mixed case pattern, and the abbrev matches this pattern as far as it goes, then the expansion is always copied verbatim, regardless of those variables. Thus, for example, if the buffer contains `variableWithSillyCasePattern`, and you type **`v a M-/`**, it copies the expansion verbatim including its case pattern.

The variable `dabbrev-abbrev-char-regexp`, if non-`nil`, controls which characters are considered part of a word, for dynamic expansion purposes. The regular expression must match just one character, never two or more. The same regular expression also determines which characters are part of an expansion. The value `nil` has a special meaning: abbreviations are made of word characters, but expansions are made of word and symbol characters.

In shell scripts and makefiles, a variable name is sometimes prefixed with `` `$' `` and sometimes not. Major modes for this kind of text can customize dynamic abbreviation to handle optional prefixes by setting the variable `dabbrev-abbrev-skip-leading-regexp`. Its value should be a regular expression that matches the optional prefix that dynamic abbreviation should ignore.

23 Miscellaneous Commands

This chapter contains several brief topics that do not fit anywhere else: reading netnews, running shell commands and shell subprocesses, using a single shared Emacs for utilities that expect to run an editor as a subprocess, printing hardcopy, sorting text, narrowing display to part of the buffer, editing double−column files and binary files, saving an Emacs session for later resumption, emulating other editors, and various diversions and amusements.

23.1 Hardcopy Output

The Emacs commands for making hardcopy let you print either an entire buffer or just part of one, either with or without page headers. See also the hardcopy commands of Dired (see section Miscellaneous File Operations) and the diary (see section Commands Displaying Diary Entries).

`M−x print−buffer`
: Print hardcopy of current buffer with page headings containing the file name and page number.

`M−x lpr−buffer`
: Print hardcopy of current buffer without page headings.

`M−x print−region`
: Like `print−buffer` but print only the current region.

`M−x lpr−region`
: Like `lpr−buffer` but print only the current region.

The hardcopy commands (aside from the Postscript 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 an option starting with `` `−' ``. For example, to specify a line width of 80 columns for all the printing you do in Emacs, set `lpr−switches` like this:

```
(setq lpr−switches '("−w80"))
```

You can specify the printer to use by setting the variable `printer−name`.

The variable `lpr−command` specifies the name of the printer program to run; the default value depends on your operating system type. On most systems, the default is `"lpr"`. The variable `lpr−headers−switches` similarly specifies the extra switches to use to make page headers. The variable `lpr−add−switches` controls whether to supply `` `−T' `` and `` `−J' `` options (suitable for `lpr`) to the printer program: `nil` means don't add them. `lpr−add−switches` should be `nil` if your printer program is not compatible with `lpr`.

23.2 Postscript Hardcopy

These commands convert buffer contents to Postscript, either printing it or leaving it in another Emacs buffer.

`M−x ps−print−buffer`
: Print hardcopy of the current buffer in Postscript form.

`M−x ps−print−region`

Print hardcopy of the current region in Postscript form.

M-x ps-print-buffer-with-faces
: Print hardcopy of the current buffer in Postscript form, showing the faces used in the text by means of Postscript features.

M-x ps-print-region-with-faces
: Print hardcopy of the current region in Postscript form, showing the faces used in the text.

M-x ps-spool-buffer
: Generate Postscript for the current buffer text.

M-x ps-spool-region
: Generate Postscript for the current region.

M-x ps-spool-buffer-with-faces
: Generate Postscript for the current buffer, showing the faces used.

M-x ps-spool-region-with-faces
: Generate Postscript for the current region, showing the faces used.

The Postscript commands, `ps-print-buffer` and `ps-print-region`, print buffer contents in Postscript form. One command prints the entire buffer; the other, just the region. The corresponding `` `-with-faces' `` commands, `ps-print-buffer-with-faces` and `ps-print-region-with-faces`, use Postscript features to show the faces (fonts and colors) in the text properties of the text being printed.

If you are using a color display, you can print a buffer of program code with color highlighting by turning on Font-Lock mode in that buffer, and using `ps-print-buffer-with-faces`.

The commands whose names have `` `spool' `` instead of `` `print' `` generate the Postscript output in an Emacs buffer instead of sending it to the printer.

23.3 Variables for Postscript Hardcopy

All the Postscript hardcopy commands use the variables `ps-lpr-command` and `ps-lpr-switches` to specify how to print the output. `ps-lpr-command` specifies the command name to run, `ps-lpr-switches` specifies command line options to use, and `ps-printer-name` specifies the printer. If you don't set the first two variables yourself, they take their initial values from `lpr-command` and `lpr-switches`. If `ps-printer-name` is `nil`, `printer-name` is used.

The variable `ps-print-header` controls whether these commands add header lines to each page--set it to `nil` to turn headers off. You can turn off color processing by setting `ps-print-color-p` to `nil`.

The variable `ps-paper-type` specifies which size of paper to format for; legitimate values include `a4`, `a3`, `a4small`, `b4`, `b5`, `executive`, `ledger`, `legal`, `letter`, `letter-small`, `statement`, `tabloid`. The default is `letter`. You can define additional paper sizes by changing the variable `ps-page-dimensions-database`.

The variable `ps-landscape-mode` specifies the orientation of printing on the page. The default is `nil`, which stands for "portrait" mode. Any non-`nil` value specifies "landscape" mode.

The variable `ps-number-of-columns` specifies the number of columns; it takes effect in both landscape and portrait mode. The default is 1.

The variable `ps-font-family` specifies which font family to use for printing ordinary text. Legitimate values include `Courier`, `Helvetica`, `NewCenturySchlbk`, `Palatino` and `Times`. The variable `ps-font-size` specifies the size of the font for ordinary text. It defaults to 8.5 points.

Many other customization variables for these commands are defined and described in the Lisp file `ps-print.el'.

23.4 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 numeric 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 numeric 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 numeric 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. Specify which field to sort by with a numeric argument: 1 to sort by field 1, etc. A negative argument means count fields from the right instead of from the left; thus, minus 1 means sort by the last field. If several lines have identical contents in the field being sorted, they keep same relative order that they had in the original buffer.

M-x sort-numeric-fields
: Like **M-x sort-fields** except the specified field is converted to an integer 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.

M-x reverse-region
: Reverse the order of the lines in the region. This is useful for sorting into descending order

by fields or columns, since those sort commands do not have a feature for doing that.

For example, if the buffer contains this:

```
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.
```

applying **`M-x sort-lines`** to the entire buffer produces this:

```
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' sorts before all lower-case letters. If you use **`C-u 2 M-x sort-fields`** instead, you get this:

```
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 of the text you want 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, as well as all the lines in between.

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 run `sort-columns`. Equivalently, you could run it with 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 Rectangles.

Many of the sort commands ignore case differences when comparing, if `sort-fold-case` is non-`nil`.

23.5 Narrowing

Narrowing means focusing in on some portion of the buffer, making the rest temporarily inaccessible. The portion which you can still get to is called the *accessible portion*. Canceling the narrowing, which makes the entire buffer once again accessible, is called *widening*. The amount of

narrowing in effect in a buffer at any time is called the buffer's *restriction*.

Narrowing can make it easier to concentrate on a single subroutine or paragraph by eliminating clutter. It can also be used to restrict the range of operation of a replace command or repeating keyboard macro.

C-x n n
: Narrow down to between point and mark (`narrow-to-region`).

C-x n w
: Widen to make the entire buffer accessible again (`widen`).

C-x n p
: Narrow down to the current page (`narrow-to-page`).

C-x n d
: Narrow down to the current defun (`narrow-to-defun`).

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 accessible part), you can't change it in any way. However, it is not gone, and if you save the file all the inaccessible text will be saved. The word `` `Narrow' `` appears in the mode line whenever narrowing is in effect.

The primary narrowing command is **C-x n n** (`narrow-to-region`). It sets the current buffer's restrictions so that the text in the current region remains accessible but all text before the region or after the region is inaccessible. Point and mark do not change.

Alternatively, use **C-x n p** (`narrow-to-page`) to narrow down to the current page. See section Pages, for the definition of a page. **C-x n d** (`narrow-to-defun`) narrows down to the defun containing point (see section Defuns).

The way to cancel narrowing is to widen with **C-x n 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 Cursor Position Information.

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; if you enable the command, confirmation will no longer be required for it. See section Disabling Commands.

23.6 Two-Column Editing

Two-column mode lets you conveniently edit two side-by-side columns of text. It uses two side-by-side windows, each showing its own buffer.

There are three ways to enter two-column mode:

F2 2 *or* **C-x 6 2**
: Enter two-column mode with the current buffer on the left, and on the right, a buffer whose name is based on the current buffer's name (`2C-two-columns`). If the right-hand buffer

doesn't already exist, it starts out empty; the current buffer's contents are not changed. This command is appropriate when the current buffer is empty or contains just one column and you want to add another column.

F2 s *or* **C-x 6 s**
: Split the current buffer, which contains two-column text, into two buffers, and display them side by side (`2C-split`). The current buffer becomes the left-hand buffer, but the text in the right-hand column is moved into the right-hand buffer. The current column specifies the split point. Splitting starts with the current line and continues to the end of the buffer. This command is appropriate when you have a buffer that already contains two-column text, and you wish to separate the columns temporarily.

F2 b ***buffer*** **RET**
C-x 6 b ***buffer*** **RET**
: Enter two-column mode using the current buffer as the left-hand buffer, and using buffer *buffer* as the right-hand buffer (`2C-associate-buffer`).

F2 s or **C-x 6 s** looks for a column separator, which is a string that appears on each line between the two columns. You can specify the width of the separator with a numeric argument to **F2 s**; that many characters, before point, constitute the separator string. By default, the width is 1, so the column separator is the character before point.

When a line has the separator at the proper place, **F2 s** puts the text after the separator into the right-hand buffer, and deletes the separator. Lines that don't have the column separator at the proper place remain unsplit; they stay in the left-hand buffer, and the right-hand buffer gets an empty line to correspond. (This is the way to write a line that "spans both columns while in two-column mode": write it in the left-hand buffer, and put an empty line in the right-hand buffer.)

The command **C-x 6 RET** or **F2 RET** (`2C-newline`) inserts a newline in each of the two buffers at corresponding positions. This is the easiest way to add a new line to the two-column text while editing it in split buffers.

When you have edited both buffers as you wish, merge them with **F2 1** or **C-x 6 1** (`2C-merge`). This copies the text from the right-hand buffer as a second column in the other buffer. To go back to two-column editing, use **F2 s**.

Use **F2 d** or **C-x 6 d** to dissociate the two buffers, leaving each as it stands (`2C-dissociate`). If the other buffer, the one not current when you type **F2 d**, is empty, **F2 d** kills it.

23.7 Saving Emacs Sessions

You can use the Desktop library to save the state of Emacs from one session to another. Saving the state means that Emacs starts up with the same set of buffers, major modes, buffer positions, and so on that the previous Emacs session had.

To use Desktop, you should use the Customization buffer (see section Easy Customization Interface) to set `desktop-enable` to a non-`nil` value, or add these lines at the end of your `` `.emacs' `` file:

```
(desktop-load-default)
(desktop-read)
```

The first time you save the state of the Emacs session, you must do it manually, with the command

M−x desktop−save. Once you have done that, exiting Emacs will save the state again−−not only the present Emacs session, but also subsequent sessions. You can also save the state at any time, without exiting Emacs, by typing **M−x desktop−save** again.

In order for Emacs to recover the state from a previous session, you must start it with the same current directory as you used when you started the previous session. This is because `desktop−read` looks in the current directory for the file to read. This means that you can have separate saved sessions in different directories; the directory in which you start Emacs will control which saved session to use.

The variable `desktop−files−not−to−save` controls which files are excluded from state saving. Its value is a regular expression that matches the files to exclude. By default, remote (ftp−accessed) files are excluded; this is because visiting them again in the subsequent session would be slow. If you want to include these files in state saving, set `desktop−files−not−to−save` to `"^$"`. See section Remote Files.

24 Dealing with Common Problems

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.

24.1 Quitting and Aborting

C-g
C-BREAK (MS-DOS)
: Quit. Cancel running or partially typed command.

C-]
: Abort innermost recursive editing level and cancel the command which invoked it (`abort-recursive-edit`).

ESC ESC ESC
: Either quit or abort, whichever makes sense (`keyboard-escape-quit`).

M-x top-level
: Abort all recursive editing levels that are currently executing.

C-x u
: Cancel a previously made change in the buffer contents (`undo`).

There are two ways of canceling commands which are not finished executing: *quitting* with **C-g**, and *aborting* with **C-]** or **M-x top-level**. Quitting cancels a partially typed command or one which is already running. Aborting exits a recursive editing level and cancels the command that invoked the recursive edit. (See section Recursive Editing Levels.)

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 in the buffer, 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 (see section Incremental Search).

On MS-DOS, the character **C-BREAK** serves as a quit character like **C-g**. The reason is that it is not feasible, on MS-DOS, to recognize **C-g** while a command is running, between interactions with the user. By contrast, it *is* feasible to recognize **C-BREAK** at all times. See section Keyboard and Mouse on MS-DOS.

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 you type it while Emacs is waiting for input.

If you quit with **C-g** a second time before the first **C-g** is recognized, you activate the "emergency escape" feature and return to the shell. See section Emergency Escape.

There may be times when you cannot quit. When Emacs is waiting for the operating system to do something, quitting is impossible unless special pains are taken for the particular system call within Emacs where the waiting occurs. We have done this for the system calls that users are likely to want to quit from, but it's possible you will find another. In one very common case--waiting for file input

or output using NFS−−Emacs itself knows how to quit, but most NFS implementations simply do not allow user programs to stop waiting for NFS when the NFS server is hung.

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 a recursive edit and type **C-u 8** to enter a numeric argument, you can cancel that argument with **C-g** and remain in the recursive edit.

The command **ESC ESC ESC** (`keyboard-escape-quit`) can either quit or abort. This key was defined because **ESC** is used to "get out" in many PC programs. It can cancel a prefix argument, clear a selected region, or get out of a Query Replace, like **C-g**. It can get out of the minibuffer or a recursive edit, like **C-]**. It can also get out of splitting the frame into multiple windows, like **C-x 1**. One thing it cannot do, however, is stop a command that is running. That's because it executes as an ordinary command, and Emacs doesn't notice it until it is ready for a command.

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 take effect 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 Recursive Editing Levels.

C-x u (`undo`) is not strictly speaking a way of canceling a command, but you can think of it as canceling a command that already finished executing. See section Undoing Changes.

25 The GNU Manifesto

The GNU Manifesto which appears below was written by Richard Stallman at the beginning of the GNU project, to ask for participation and support. For the first few years, it was updated in minor ways to account for developments, but now it seems best to leave it unchanged as most people have seen it.

Since that time, we have learned about certain common misunderstandings that different wording could help avoid. Footnotes added in 1993 help clarify these points.

For up−to−date information about the available GNU software, please see the latest issue of the GNU's Bulletin. The list is much too long to include here.

25.1 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.(5) 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 file names, file version numbers, a crashproof file system, file name 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.

25.2 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.

25.3 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.

25.4 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.

25.5 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.

25.6 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.

25.7 Why All Computer Users Will Benefit

Once GNU is written, everyone will be able to obtain good system software free, just like air.(6)

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.

25.8 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.(7)

We must distinguish between support in the form of real programming work and mere handholding. 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 distribution 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?(8)

> "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 benefiting 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.(9)

"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.

26 Footnotes

(1)

In some editors, search−and−replace operations are the only convenient way to make a single change in the text. **(2)**

Some systems use **`Mouse-3`** for a mode−specific menu. We took a survey of users, and found they preferred to keep **`Mouse-3`** for selecting and killing regions. Hence the decision to use **`C-Mouse-3`** for this menu.
(3)

You should not suspend the shell process. Suspending a subjob of the shell is a completely different matter−−that is normal practice, but you must use the shell to continue the subjob; this command won't do it.
(4)

This dissociword actually appeared during the Vietnam War, when it was very appropriate.
(5)

The wording here was careless. The intention was that nobody would have to pay for *permission* to use the GNU system. But the words don't make this clear, and people often interpret them as saying that copies of GNU should always be distributed at little or no charge. That was never the intent; later on, the manifesto mentions the possibility of companies providing the service of distribution for a profit. Subsequently I have learned to distinguish carefully between "free" in the sense of freedom and "free" in the sense of price. Free software is software that users have the freedom to distribute and change. Some users may obtain copies at no charge, while others pay to obtain copies−−and if the funds help support improving the software, so much the better. The important thing is that everyone who has a copy has the freedom to cooperate with others in using it.
(6)

This is another place I failed to distinguish carefully between the two different meanings of "free". The statement as it stands is not false−−you can get copies of GNU software at no charge, from your friends or over the net. But it does suggest the wrong idea.
(7)

Several such companies now exist.
(8)

The Free Software Foundation raises most of its funds from a distribution service, although it is a charity rather than a company. If *no one* chooses to obtain copies by ordering from the FSF, it will be unable to do its work. But this does not mean that proprietary restrictions are justified to force every user to pay. If a small fraction of all the users order copies from the FSF, that is sufficient to keep the FSF afloat. So we ask users to choose to support us in this way. Have you done your part?
(9)

A group of computer companies recently pooled funds to support maintenance of the GNU C Compiler.

This document was generated on 6 November 1998 using the texi2html translator version 1.52.