GNU Emacs Manual

GNU Emacs Manual

Ninth Edition, Emacs version 19.22

for Unix Users

November 1993

Richard Stallman

Ninth Edition
For Emacs Version 19.19,
Printed August, 1993.

ISBN 1-882114-03-5

Published by the Free Software Foundation
675 Massachusetts Avenue
Cambridge, MA 02139 USA

Cover art by Etienne Suvasa.

Preface

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

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

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

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

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

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

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

Distribution

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

The easiest way to get a copy of GNU Emacs is from someone else who has it. You need not ask for permission to do so, or tell any one else; just copy it. If you have access to the Internet, you can get the latest distribution version of GNU Emacs from host '`prep.ai.mit.edu`' using anonymous login. See the file '`/pub/gnu/GETTING.GNU.SOFTWARE`' on that host to find out about your options for copying and which files to use.

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

You can also order copies of GNU Emacs from the Free Software Foundation, on various magnetic media or 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 at the end of manuals printed by the Foundation. It is also included in the file '`etc/ORDERS`' in the Emacs distribution. For further information, write to

Free Software Foundation
675 Mass Ave
Cambridge, MA 02139
USA

The income from distribution fees goes to support the foundation's purpose: the development of 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. If you use GNU Emacs at your workplace, suggest that the company make a donation. If the management of your office opposes the idea of donations, you might instead suggest ordering a CD-ROM from the Foundation occasionally.

GNU GENERAL PUBLIC LICENSE

Version 2, June 1991

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.

TERMS AND CONDITIONS FOR COPYING, DISTRIBUTION AND MODIFICATION

0. 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.

1. 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.

2. 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:

   a. You must cause the modified files to carry prominent notices stating that you changed the files and the date of any change.

   b. 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.

c. 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.

3. 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:

   a. 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,

   b. 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,

   c. 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.

4. 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.
5. 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.
6. 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.
7. 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.

8. 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.

9. 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.

10. 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

11. 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.
12. IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MAY MODIFY AND/OR REDISTRIBUTE THE PROGRAM AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE PROGRAM (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE PROGRAM TO OPERATE WITH ANY OTHER PROGRAMS), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

END OF TERMS AND CONDITIONS

How to Apply These Terms to Your New Programs

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

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

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

This program is free software; you can redistribute it and/or
modify it under the terms of the GNU General Public License
as published by the Free Software Foundation; either version 2
of the License, or (at your option) any later version.

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

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

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

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

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

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

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

```
Yoyodyne, Inc., hereby disclaims all copyright
interest in the program `Gnomovision'
(which makes passes at compilers) written
by James Hacker.

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

This General Public License does not permit incorporating your program into proprietary programs. If your program is a subroutine library, you may consider it more useful to permit linking proprietary applications with the library. If this is what you want to do, use the GNU Library General Public License instead of this License.

Introduction

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

We say that Emacs is a *display* editor because normally the text being edited is visible on the screen and is updated automatically as you type your commands. See Chapter 1 [Screen], page 15.

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 Chapter 4 [Basic Editing], page 25.

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

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 Chapter 8 [Help], page 45.

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 22.7.2 [Comments], page 175). Another sort of customization is rearrangement of the command set. For example, if you prefer the four basic cursor motion commands (up, down, left and right) on keys in a diamond pattern on the keyboard, you can have it. See Chapter 31 [Customization], page 291.

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

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 them, and edit files at the same time as you run shell commands.

1 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 under X you can create additional frames if you wish. See Chapter 18 [Frames], page 129.

When you start Emacs, the entire frame except for the last line is devoted to the text you are editing. This area is called *window*. The last line is a special *echo area* or *minibuffer window* where prompts appear and where you can enter responses. 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 Chapter 17 [Windows], page 125). 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. The other windows display text for reference only, unless/until you select them.

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

1.1 Point

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

While the cursor appears to point *at* a character, 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, 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.

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

1.2 The Echo Area

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

Echoing means printing out the characters that you type. 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 12.6 [Display Vars], page 73).

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.

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

1.3 The Mode Line

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

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

Normally, the mode line looks like this:

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

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

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

buf is the name of the window's *buffer*. In most cases this is the same as the name of a file you are editing. See Chapter 16 [Buffers], page 119.

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.

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, and C mode. See Chapter 19 [Major Modes], page 137, for details of how the modes differ and how to select one.

minor is a list of some of the *minor modes* that are turned on at the moment in the window's chosen buffer. '`Fill`' means that Auto Fill mode is on. '`Abbrev`' means that Word Abbrev mode is on. '`Ovwrt`' means that Overwrite mode is on. See Section 31.1 [Minor Modes], page 291, 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 30.4 [Narrowing], page 282. '`Def`' means that a keyboard macro is being defined. See Section 31.3 [Keyboard Macros], page 298.

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

In addition, if Emacs is currently inside a recursive editing level, square brackets ('`[...]`') appear around the parentheses that surround the modes. If Emacs is in one recursive editing level within another, double square brackets appear, and so on. Since recursive editing levels affect Emacs globally and not any one buffer, the square brackets appear in every window's mode line or not in any of them. See Section 30.9 [Recursive Edit], page 287.

See Section 12.5 [Optional Display], page 72, for features that add other handy information to the mode line, such as the current line number of point, the current time, and whether mail has arrived for you.

2 Characters, Keys and Commands

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

2.1 Keyboard Input

GNU Emacs uses an extension of the ASCII character set for keyboard input. 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 and then pressing `a`.

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

On ASCII terminals, the shift key is meaningless with control characters: `C-a` and `C-A` are the same character, and Emacs cannot distinguish them. Under X Windows, these are distinct characters, but the standard key bindings treat them the same in all contexts.

On ASCII terminals, there are only 32 possible control characters. These are the control variants of letters and '`@[]\^_`'. Under X Windows, every non-control character has a control variant. For example, `C-+` and `C-5` are meaningful under X.

Emacs extends the ASCII character code by adding an extra bit to each character. The additional bit is called Meta. Any character can be made Meta; examples of Meta characters 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 traditional reasons, `M-C-a` is usually called `C-M-a`; logically speaking, the order in which the modifier keys `CTRL` and `META` are mentioned does not matter.

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

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

X Windows provides several other modifier keys that can be applied to any keyboard input character. These are called `SUPER`, `HYPER` and `ALT`. With them, you can make characters that we denote with '`s-`', '`H-`' and '`A-`'. Thus, `s-H-C-x` is short for `Super-Hyper-Control-x`. Not all X terminals actually provide keys for these modifier flags, and the standard key bindings of Emacs do not include such characters. 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. These inputs do not have numeric character codes. Instead, Emacs represents them by their names (actually, Lisp objects called *symbols*).

Input characters and non-character inputs are collectively called *input events*.

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 converts them to names before any other part of Emacs gets to see them.

2.2 Keys

A *key sequence* (*key*, for short) is a sequence of input events that combine as part of the invocation of a single command. Recall that input events include both keyboard characters and non-character inputs (function keys, arrow keys, mouse buttons, and so forth).

If the sequence is enough to invoke a command, it is a *complete key*. If it isn't long enough to be complete, we call it a *prefix 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`.

Most single characters constitute complete keys in the standard Emacs command bindings. A few of them are prefix keys. A prefix key can be followed by additional input characters (or other events) to make a longer key, which may itself be complete or a prefix.

For example, `C-x` is a prefix key, so `C-x` and the next input character 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 any key sequence longer than one character must be reached through a chain of prefix keys.

By contrast, the two-character sequence `C-f C-k` is not a key, because the `C-f` is a complete key in itself. It's impossible to give `C-f C-k` an independent meaning as a command. `C-f C-k` is two key sequences, not one.

All told, the prefix keys in Emacs are `C-c`, `C-x`, `C-h`, `C-x C-a`, `C-x n`, `C-x r`, `C-x v`, `C-x 4`, `C-x 5`, and `ESC`. But this is not cast in concrete; it is just a matter of Emacs's standard key bindings. In customizing Emacs, you could make new prefix keys, or eliminate these. See Section 31.4 [Key Bindings], page 302.

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

Typing the help character (`C-h`) after a prefix character usually displays a list of the commands starting with that prefix. There are a few prefix characters for which this doesn't work—for historical reasons, they have other meanings for `C-h` which are not easy to change.

2.3 Keys and Commands

This manual is full of passages that tell you what particular keys do. But Emacs does not assign meanings to keys directly. Instead, Emacs assigns meanings to 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 31.4.1 [Keymaps], page 302.

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 customizer the information he needs, we state the name of the command which really does the work in parentheses after mentioning the key that runs it. For example, we will say that "The command `C-n` (`next-line`) moves point vertically down," meaning that `next-line` is a command that moves vertically down and `C-n` is a key that is standardly bound to it.

While we are on the subject of information for customization only, it's a good time to tell you about *variables*. Often the description of a command will say, "To change this, set the variable `mumble-foo`." A variable is a name used to remember a value. Most of the variables documented in this manual exist just to facilitate customization: some command or other part of Emacs examines the variable and behaves differently accordingly. Until you are interested in customizing, you can ignore the information about variables. When you are ready to be interested, read the basic information on variables, and then the information on individual variables will make sense. See Section 31.2 [Variables], page 292.

2.4 Character Set for Text

Emacs buffers use an 8-bit character set, because bytes have 8 bits. ASCII graphic characters in Emacs buffers are displayed with their graphics. The newline character (which has the same character code as `LFD`) is displayed by starting a new line. The tab character is displayed by moving to the next tab stop column (usually every 8 columns). Other control characters are displayed as a caret ('`^`') followed by the non-control version of the character; thus, `C-a` is displayed as '`^A`'.

Non-ASCII characters 128 and up are displayed with octal escape sequences; thus, character code 243 (octal) is displayed as '`\243`'. You can customize the display of these character codes (or ANSI characters) by creating a *display table*; this is useful for editing files that use 8-bit European character sets. See section "Display Tables" in *The Emacs Lisp Reference Manual*.

3 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. On a window system, Emacs opens a window of its own. You can begin typing Emacs commands immediately afterward.

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

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 it to run other shell commands while Emacs operates its own X windows.

When Emacs starts up, it makes a buffer named '`*scratch*`'. That's the buffer you start out in. The '`*scratch*`' 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 31.7 [Init File], page 311.)

It is also possible to specify files to be visited, Lisp files to be loaded, and functions to be called, by giving Emacs arguments in the shell command line. See Appendix A [Command Arguments], page 331. 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.

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.

3.1 Exiting Emacs

There are two commands for exiting Emacs because there are two kinds of exiting: *suspending* Emacs and *killing* Emacs.

Suspending means stopping Emacs temporarily and returning control to its parent process (usually a shell), allowing you to resume editing later in the same Emacs job, with the same files, same kill ring, same undo history, and so on. This is the usual way to exit.

Killing Emacs means destroying the Emacs job. You can run Emacs again later, but you will get a fresh Emacs; there is no way to resume the same editing session after it has been killed.

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

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

To suspend Emacs, type `C-z` (`suspend-emacs`). This takes you back to the shell from which you invoked Emacs. You can resume Emacs with the shell command '`%emacs`' in most common shells.

On systems that do not permit programs to be suspended, `C-z` runs an inferior shell that communicates directly with the terminal, and Emacs waits until you exit the subshell. (The 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.

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-frame`, which temporarily closes up the selected Emacs frame. 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. Unless a numeric argument is used, this command first offers to save any modified buffers. If you do not save them all, it asks for reconfirmation with `yes` before killing Emacs, since any changes not saved 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.

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 (see Section 31.4.1 [Keymaps], page 302).

4 Basic Editing Commands

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

To clear the screen and redisplay, type `C-l` (`recenter`).

4.1 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 1.1 [Point], page 15). 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 mode called *Auto Fill* mode. See Section 21.5 [Filling], page 147, for how to use Auto Fill mode.

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.

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 three octal digits inserts the character with the specified character code.

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

If you prefer to have text characters replace (overwrite) existing text rather than shove it to the right, you can enable Overwrite mode, a minor mode. See Section 31.1 [Minor Modes], page 291.

4.2 Changing the Location of Point

To do more than insert characters, you have to know how to move point (see Section 1.1 [Point], page 15). The simplest way to do this is with arrow keys or the left mouse button.

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 how many 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 4.9 [Arguments], page 31, for more information on numeric arguments.

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

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

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

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

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

If you set the variable `track-eol` to a non-`nil` value, then `C-n` and `C-p` when at the end of the starting line move to the end of another line. Normally, `track-eol` is `nil`. See Section 31.2 [Variables], page 292, 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).

4.3 Erasing Text

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

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

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

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

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

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

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

If you delete or kill text by mistake, you can use the undo command to get it back. See Chapter 5 [Undo], page 33.

See Section 10.1 [Killing], page 57, for more flexible ways of killing text.

4.4 Files

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

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

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

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

Emacs obeys the command by *visiting* the file: creating a buffer, copying the contents of the file into the buffer, and then displaying the buffer for you to edit. Then you can make changes, and *save* the file by typing `C-x C-s` (`save-buffer`). This makes the changes permanent by copying the altered contents of the buffer back into the file '`/usr/rms/foo.c`'. Until 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 Chapter 15 [Files], page 93.

4.5 Help

If you forget what a key does, you can find out with the Help character, which is `C-h`. Type `C-h k` followed by the key you want to know about; for example, `C-h k C-n` tells you all about what `C-n` does. `C-h` is a prefix key; `C-h k` is just one of its subcommands (the command `describe-key`). The other subcommands of `C-h` provide different kinds of help. Type `C-h` three times to get a description of all the help facilities. See Chapter 8 [Help], page 45.

4.6 Blank Lines

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

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

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

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

You can make several blank lines by typing `C-o` several times, or by giving it a numeric argument to tell it how many blank lines to make. See Section 4.9 [Arguments], page 31, 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 21.5.3 [Fill Prefix], page 149.

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.

4.7 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 21.5 [Filling], page 147) if that's what you want.

Instead of continuation, you 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.

You can turn off continuation for a particular buffer by setting the variable `truncate-lines` to non-`nil` in that buffer. (See Section 31.2 [Variables], page 292.) Truncation instead of continuation also happens whenever horizontal scrolling is in use, and optionally whenever side-by-side windows are in use (see Chapter 17 [Windows], page 125). 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 31.2.4 [Locals], page 295.

See Section 12.6 [Display Vars], page 73, for additional variables that affect how text is displayed.

4.8 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`).

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

There are two commands for printing the current line number. `M-x what-line` computes the current line number and displays it in the echo area. `M-x line-number-mode` enables display of the current line number in the mode line; once you turn this on, the number updates as you move point, so it remains valid all the time. See Chapter 2 [Mode Line], page 19.

Line numbers count from one at the beginning of the buffer. To go to a given line by number, use `M-x goto-line`; it prompts you for the line number.

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 21.4 [Pages], page 146.

While on this subject, we might as well mention `M-=` (`count-lines-region`), which prints the number of lines in the region (see Chapter 9 [Mark], page 51). See Section 21.4 [Pages], page 146, for the command `C-x l` which counts the lines in the current page.

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

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

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

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

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

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

If the buffer has been narrowed, making some of the text at the beginning and the end temporarily off limits, `C-x =` prints additional text describing the currently accessible range. For example, it might display this:

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

where the two extra numbers give the smallest and largest character position that point is allowed to assume. The characters between those two positions are the accessible ones. See Section 30.4 [Narrowing], page 282.

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 looks like this:

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

4.9 Numeric Arguments

Any Emacs command can be given a *numeric argument* (also called a *prefix argument*). Some commands interpret the argument as a repetition count. For example, giving an argument of ten to the key `C-f` 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 the `META` key. For example,

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

would move down five lines. The characters `Meta-1`, `Meta-2`, and so on, as well as `Meta--`, do this because they are keys bound to commands (`digit-`

`argument` and `negative-argument`) that are defined to contribute to an argument for the next command.

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

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

Some commands care only about whether there is an argument, and not about its value. For example, the command `M-q` (`fill-paragraph`) with no argument fills text; with an argument, it justifies the text as well. (See Section 21.5 [Filling], page 147, for more information on `M-q`.) Just `C-u` is a handy way of providing an argument for such commands.

Some commands use the value of the argument as a repeat count, but do something peculiar when there is no argument. For example, the command `C-k` (`kill-line`) with argument *n* kills *n* lines, including their terminating newlines. But `C-k` with no argument is special: it kills the text up to the next newline, or, if point is right at the end of the line, it kills the newline itself. Thus, two `C-k` commands with no arguments can kill a nonblank line, just like `C-k` with an argument of one. (See Section 10.1 [Killing], page 57, for more information on `C-k`.)

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

You can use a numeric argument to insert multiple copies of a character. This is straightforward unless the character is a digit. To prevent the digit from becoming part of the argument, type another `C-u`. That terminates the argument. If you then type another digit, then the digit acts as a self-inserting character and uses the argument as a repeat count.

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.

5 Undoing Changes

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

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

`C-_` The same.

The command `C-x u` or `C-_` is how you undo. The first time you give this command, it undoes the last change. Point moves back to where it was before the command that made the change.

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

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

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

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

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

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 Chapter 10 [Mark Ring], page 57).

When the undo information of 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 harsher 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 in the same fashion on any terminal.

6 The Minibuffer

The *minibuffer* is the facility used by Emacs commands to read arguments more complicated than a single number. Minibuffer arguments can be file names, buffer names, Lisp function names, Emacs command names, Lisp expressions, and many other things, depending on the command reading the argument. 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 (e.g., just type `RET`). For example, commands that read buffer names always show a default, which is the name of the buffer that will be used if you type just `RET`.

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

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

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

6.1 Minibuffers for File Names

Sometimes the minibuffer starts out with text in it. For example, when you are supposed to give a file name, the minibuffer starts out containing the *default directory*, which ends with a slash. This is to inform you which directory the file will be found in if you do not specify a directory. For example, the minibuffer might start out with

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

where '`Find File: `' is the prompt. Typing `buffer.c` specifies the file '`/u2/emacs/src/buffer.c`'. To find files in nearby directories, use `..`; thus, if you type `../lisp/simple.el`, 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 21.1 [Words], page 143).

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

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

6.2 Editing in the Minibuffer

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

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

The minibuffer has its own window which always has space on the screen but acts as if it were not there when the minibuffer is not in use. When the minibuffer is in use, its window is just like the others; you can switch to another window with `C-x o`, edit text in other windows and perhaps even visit more files, before returning to the minibuffer to submit the argument. You can kill text in another window, return to the minibuffer window, and then yank the text to use it in the argument. See Chapter 17 [Windows], page 125.

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

If while in the minibuffer you issue a command that displays help text of any sort in another window, 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 17.3 [Other Window], page 126.

You can't use a command that reads a minibuffer argument while you are in the minibuffer window. 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.

6.3 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.

6.3.1 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 only two: `auto-fill-mode` and `auto-save-mode`. These are the same as far as `auto-`, so the '`au`' in the minibuffer changes to '`auto-`'.

If you type `TAB` again immediately, there are multiple possibilities for the very next character—it could be '`s`' or '`f`'—so no more characters are added; 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.

6.3.2 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).

Mouse-2
: Clicking mouse button 2 on a completion in the list of possible completions chooses that completion (mouse-choose-completion). Like the above commands, this command is used while you are in the minibuffer; but click in the list of completions, not in the minibuffer itself.

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.

6.3.3 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 17.3 [Other Window], page 126).

6.3.4 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.

6.4 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.

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

`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, type `M-p` (`previous-history-element`) to "move to" the next earlier minibuffer input, and use `M-n` (`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.

There are also commands to search forward or backward through the history. As of this writing, 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.

All uses of the minibuffer record your input on a history list, but there are separate history lists for different kinds of input. For example, there is a list for file names, used by all the commands that read file names. There is a list for arguments of commands like `query-replace`. There are also very specific history lists, including one for command names read by `M-X` and one for compilation commands read by `compile`. Finally, there is one "miscellaneous" history list that most minibuffer arguments use.

6.5 Repeating Minibuffer Commands

Every command that uses the minibuffer at least once is recorded on a special history list, together with the values of their arguments, so that you can repeat the entire command. In particular, every use of `M-x` is recorded, 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 6.4 [Minibuffer History], page 39) 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 reexecute a command by calling `eval` with the `command-history` element.

7 Running Commands by Name

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

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

The way to run a command by name is to start with `M-x`, type the command name, and finish it with `RET`. `M-x` uses the minibuffer to read the command name. `RET` exits the minibuffer and runs the command. The string '`M-x`' appears at the beginning of the minibuffer as a *prompt* to remind you to enter the name of a command to be run. See Chapter 6 [Minibuffer], page 35, for full information on the features of the minibuffer.

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

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

or

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

Note that `forward-char` is the same command that you invoke with the key `C-f`. You can run any Emacs command by its name using `M-x`, whether or not any keys are bound to it.

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

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

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

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

8 Help

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

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

```
a b c f i k l m n p s t v w C-c C-d C-n C-w.  Type C-h for more help:
```

You should then type one of those characters.

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

`C-h` means "help" in various other contexts as well. For example, in `query-replace`, it describes the options available.

Here is a summary of the defined help commands.

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

`C-h b`
: Display a table of all key bindings in effect now, in this order: minor mode bindings, major mode bindings, and global bindings (`describe-bindings`).

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

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

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

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

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

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

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

`C-h p` Find packages by topic keyword (`finder-by-keyword`).

`C-h s` Display current contents of the syntax table, plus an explanation of what they mean (`describe-syntax`). See Section 31.6 [Syntax], page 311.

`C-h t` Enter the Emacs interactive tutorial (`help-with-tutorial`).

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

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

`C-h C-f` *function* `RET`
: Enter Info and go to the node documenting the Emacs function *function* (`info-goto-emacs-command-node`).

`C-h C-k` *key*
: Enter Info and go to the node where the key sequence *key* is documented (`info-goto-emacs-key-command-node`).

8.1 Documentation for a Key

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

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

`C-h c` and `C-h k` work for any sort of key sequences, including function keys and mouse events.

8.2 Help by Command or Variable Name

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

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

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

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

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

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

`C-h w` *command* `RET` tells you what keys are bound to *command*. It prints a list of the keys in the echo area. If it says the command is not on any key, you must use `M-x` to run it.

`C-h v` (`describe-variable`) is like `C-h f` but describes Lisp variables instead of Lisp functions. Its default is the Lisp symbol around or before point, but only if that is the name of a known Lisp variable. See Section 31.2 [Variables], page 292.

8.3 Apropos

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

Because `C-h a` looks only for functions whose names contain the string which you specify, you must use ingenuity in choosing the string. If you are looking for commands for killing backwards and `C-h a kill-backwards RET` doesn't reveal any, don't give up. Try just `kill`, or just `backwards`, or just `back`. Be persistent. Also note that you can use a regular expression as the argument, for more flexibility (see Section 13.5 [Regexps], page 79).

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

> char, line, word, sentence, paragraph, region, page, sexp, list, defun, rect, buffer, frame, window, file, dir, register, mode, beginning, end, forward, backward, next, previous, up, down, search, goto, kill, delete, mark, insert, yank, fill, indent, case, change, set, what, list, find, view, describe.

To list all Lisp symbols that contain a match for a regexp, not just the ones that are defined as commands, use the command `M-x apropos` instead of `C-h a`. This command does not check key bindings by default; specify a numeric argument if you want it to check them.

The `super-apropos` command is like `apropos` except that it searches documentation strings as well as symbol names for matches for the specified regular expression.

8.4 Keyword Search for Lisp Libraries

The `C-h p` command lets you search the standard Emacs Lisp libraries by topic keywords. Here is a partial list of keywords you can use:

'`abbrev`' Abbreviation handling, typing shortcuts, macros.

'`bib`' Support for the bibliography processor `bib`.

'`c`' C and C`++` language support.

'`calendar`'
Calendar and time management support.

'`comm`' Communications, networking, remote access to files.

'`docs`' Support for Emacs documentation.

'`emulations`'
Emulations of other editors.

'`extensions`'
Emacs Lisp language extensions.

'`faces`' Support for using faces (fonts and colors; see Section 18.7 [Faces], page 134).

'games' Games, jokes and amusements.

'hardware'
Support for interfacing with exotic hardware.

'help' Support for on-line help systems.

'i18n' Internationalization and alternate character-set support.

'internal'
Code for Emacs internals, build process, defaults.

'languages'
Specialized modes for editing programming languages.

'lisp' Support for using Lisp (including Emacs Lisp).

'local' Libraries local to your site.

'maint' Maintenance aids for the Emacs development group.

'mail' Modes for electronic-mail handling.

'news' Support for netnews reading and posting.

'non-text'
Support for editing files that are not ordinary text.

'processes'
Process, subshell, compilation, and job control support.

'terminals'
Support for terminal types.

'tex' Support for the TeX formatter.

'tools' Programming tools.

'unix' Front-ends/assistants for, or emulators of, Unix features.

'vms' Support code for VMS.

'wp' Word processing.

8.5 Other Help Commands

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

There are two special help commands for accessing Emacs documentation through Info. `C-h C-f` *function* `RET` enters Info and goes straight to the documentation of the Emacs function *function*. `C-h C-k` *key* enters

Info and goes straight to the documentation of the key *key*. These two keys run the commands `Info-goto-emacs-command-node` and `Info-goto-emacs-key-command-node`.

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

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

`C-h b` (`describe-bindings`) and `C-h s` (`describe-syntax`) present other information about the current Emacs mode. `C-h b` displays a list of all the key bindings now in effect; the local bindings defined by the current minor modes first, then the local bindings defined by the current major mode, and finally the global bindings (see Section 31.4 [Key Bindings], page 302). `C-h s` displays the contents of the syntax table, with explanations of each character's syntax (see Section 31.6 [Syntax], page 311).

You can get a similar list for a particular prefix key by typing `C-h` after the prefix key. (There are a few prefix keys for which this does not work—those that provide their own bindings for `C-h`. One of these is `ESC`, because `ESC C-h` is actually `C-M-h`, which marks a defun.)

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

9 The Mark and the Region

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

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

Many commands that insert text, such as `C-y` (`yank`) and `M-x insert-buffer`, position 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*.

9.1 Setting the Mark

Here are some commands for setting the mark:

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

`C-@` The same.

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

For example, suppose you wish to convert part of the buffer to all 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.

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.

`C-x C-x` is also useful when you are satisfied with the position of point but want to move the mark; 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`.

9.2 Transient Mark Mode

Many Emacs commands move the mark and invisibly set new regions. This means that there is almost always some region that you can act on. This is convenient, provided you get used to keeping track of the mark's position.

Some people prefer a more rigid mode of operation in which you must set up a region for each command that uses one—in which the region "lasts" only temporarily. This is called Transient Mark mode. It is particularly well-suited to window systems such as X, since Emacs can highlight the region when it is active.

To enable Transient Mark mode, type `M-x transient-mark-mode`. This command toggles the mode, so you can issue it again to return to the normal Emacs way of handling the mark and the region.

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 change in extent.
- On a window system, an easy way to select a region is to press the button `MOUSE-1` (normally the left button) at one end of it, drag the mouse to the other end, and then release the button.
- When the mark is active, you can execute any commands you want on the region, such as killing, indentation, 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->` that "leave the mark behind" do not activate the new mark. You can activate the new region by executing `C-x C-x` (`exchange-point-and-mark`).
- Quitting with `C-g` deactivates the mark.

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.

When multiple windows show the same buffer, they can have different regions, because they can have different values of point. In Transient Mark mode, each window highlights its own region. The part that is highlighted in the selected window is the region that editing commands use. See Chapter 17 [Windows], page 125.

9.3 Operating on the Region

Once you have set up a region and the mark is active, you can do many things to the text in it:

- Kill it with `C-w` (see Section 10.1 [Killing], page 57).
- Save it in a register with `C-x r s` (see Chapter 11 [Registers], page 67).
- Save it in a buffer or a file (see Section 10.3 [Accumulating Text], page 62).
- Convert case with `C-x C-l` or `C-x C-u` (see Section 21.6 [Case], page 151).
- Indent it with `C-x TAB` or `C-M-\` (see Chapter 20 [Indentation], page 139).
- Fill it as text with `M-x fill-region` (see Section 21.5 [Filling], page 147).
- Print hardcopy with `M-x print-region` (see Section 30.5 [Hardcopy], page 283).
- Evaluate it as Lisp code with `M-x eval-region` (see Section 23.5 [Lisp Eval], page 206).

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

9.4 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 21.3 [Paragraphs], page 145). 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 22.4 [Defuns], page 166). `C-x C-p` (`mark-page`) puts point before the current page, and mark at the end (see Section 21.4 [Pages], page 146). 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.

9.5 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 11.1 [RegPos], page 67).

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` circulates through the positions currently in the ring.

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

10 Killing and Moving Text

Killing means erasing text and copying it into the *kill ring*, from which it can be retrieved by *yanking* it. Some more recent 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 elsewhere it 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.

10.1 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 Chapter 5 [Undo], page 33).

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.

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.

10.1.1 Deletion

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

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

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

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

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

`M-^` Join two lines by deleting the intervening newline, 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 Chapter 20 [Indentation], page 139.

10.1.2 Killing by Lines

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

The simplest kill command is `C-k`. If given at the beginning of a line, it kills all the text on the line, leaving it blank. 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`.

10.1.3 Other Kill Commands

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

`M-d` Kill word (`kill-word`). See Section 21.1 [Words], page 143.

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

`C-x DEL` Kill back to beginning of sentence (`backward-kill-sentence`). See Section 21.2 [Sentences], page 144.

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

`C-M-k` Kill sexp (`kill-sexp`). See Section 22.2 [Lists], page 164.

`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 21.1 [Words], page 143); sexps, with `C-M-k` (see Section 22.2 [Lists], page 164); and sentences, with `C-x DEL` and `M-k` (see Section 21.2 [Sentences], page 144).

10.2 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`).

10.2.1 The Kill Ring

All killed text is recorded in the *kill ring*, a list of blocks of text that have been killed. There is only one kill ring, used in all buffers, so you can kill text in one buffer and yank it in another buffer. This is the usual way to move text from one file to another. (See Section 10.3 [Accumulating Text], page 62, for some other ways.)

The command `C-y` (`yank`) reinserts the text of the most recent kill. It leaves the cursor at the end of the text. It sets the mark at the beginning of the text. See Chapter 9 [Mark], page 51.

`C-u C-y` leaves the cursor in front of the text, and sets the mark after it. This is only if the argument is specified with just a `C-u`, precisely. Any other sort of argument, including `C-u` and digits, specifies an earlier kill to yank (see Section 10.2.3 [Earlier Kills], page 61).

If you wish to copy a block of text, you might want to 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.

10.2.2 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` gets it all back 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 is a line ⋆of sample text.
```

with point as shown. 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.

10.2.3 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`.

10.3 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 Chapter 11 [Registers], page 67). 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, them 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—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 `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 Chapter 16 [Buffers], page 119, 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.

10.4 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 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 delete-rectangle`
Delete the text of the region-rectangle without saving it in any special way.

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

`M-x string-rectangle RET` *string* `RET`
Insert *string* on each line of the region-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 `M-x 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.

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 point one line vertically down, and so on. The number of lines affected is determined by the height of the saved rectangle.

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

You can also copy rectangles into and out of registers with `C-x r r` *r* and `C-x r i` *r*. See Section 11.3 [Rectangle Registers], page 68.

The command `M-x string-rectangle` is similar to `C-x r o`, but it inserts a specified string instead of blanks. You specify the string with the minibuffer. Since the length of the string specifies how many columns to insert, the width of the region-rectangle does not matter for this command. What does matter is the position of the left edge (which specifies the column position for the insertion in each line) and the range of lines that the rectangle occupies. The previous contents of the text after the insertion column are pushed rightward.

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

11 Registers

Emacs *registers* are places you can save text or positions for later use. Text and rectangles saved in registers can be copied into the buffer once or many times; you can move point to a position saved in a register.

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.

11.1 Saving Positions in Registers

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

`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 same position using the same register any number of times.

11.2 Saving Text in Registers

When you want to insert a copy of the same piece of text 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 with `C-x r s` (`copy-to-register`) and then retrieve it with `C-x r i` (`insert-register`).

`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.

11.3 Saving Rectangles in Registers

A register can contain a rectangle instead of linear text. The rectangle is represented as a list of strings. See Section 10.4 [Rectangles], page 63, 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-region-to-rectangle`). 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.

11.4 Saving Window Configurations in Registers

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

11.5 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.)

12 Controlling the Display

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

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

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

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

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

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

The names of all scroll commands are based on the direction that the text moves in the window. Thus, the command to scrolling forward is called `scroll-up`, since the text moves up.

12.1 Scrolling

If a buffer contains text that is too large to fit entirely within a window that is displaying the buffer, Emacs shows a contiguous 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.

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`.

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

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

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. Usually the scrolling is done so as to put point vertically centered within the window. However, if the variable `scroll-step` has a nonzero value, an attempt is made to scroll the buffer by that many lines; if that is enough to bring point back into visibility, that is what is done.

12.2 Horizontal Scrolling

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

The command `C-x <` (`scroll-left`) scrolls the selected window to the left by *n* columns with argument *n*. 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 displaying 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 normally display.

12.3 Selective Display

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

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

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

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

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

12.4 European Character Set Display

Emacs can display accented characters, assuming the font in use supports them. The `M-x standard-display-european` command toggles European character display. When enabled, Emacs displays characters according to the ISO 8859 Latin-1 encoding for European languages; when disabled, Emacs uses octal notation for all characters not specified by the ordinary ASCII standard. Load the library `iso-syntax` to specify the correct syntactic properties for the character codes 128 and above.

With a prefix argument, `M-x standard-display-european` enables European character display if and only if the argument is positive.

If your terminal can send character codes 128 and up to represent ISO Latin-1 characters, execute the following expression to enable Emacs to understand them:

```
(set-input-mode (car (current-input-mode))
                (nth 1 (current-input-mode))
                0)
```

Otherwise, you can load the library `iso-transl` to turn the key `C-x 8` into a "compose character" prefix for entry of the extra ISO Latin-1 printing characters. `C-x 8` is good for insertion (in the minibuffer as well as other buffers), for searching, and in any other context where a key sequence is allowed.

If you enter non-ASCII ISO Latin-1 characters often, you might find ISO Accents mode convenient. When this minor mode is enabled, the characters ‘`` ` ``’, ‘`'`’, ‘`"`’, ‘`^`’, ‘`/`’ and ‘`~`’ modify the following letter by adding the corresponding diacritical mark to it, if possible. To enter one of those six special characters, type a space after it.

Some of those characters have a corresponding "dead key" accent character in the ISO Latin-1 character set; to enter that character, type the corresponding ASCII character twice. For example, `''` enters the Latin-1 character acute-accent (character code 0264).

In addition to the accented letters, you can use these special sequences in ISO Accents mode to enter certain other ISO Latin-1 characters:

`~C` ‘`C`’ with cedilla.

`~D` ‘`D`’ with stroke.

`~c` ‘`c`’ with cedilla.

`~d` ‘`d`’ with stroke.

`~<` Left guillemet.

`~>` Right guillemet.

This feature is available whenever a key sequence is expected: for ordinary insertion, for searching, and for certain command arguments.

To enable or disable ISO Accents mode, use the command `M-x iso-accents-mode`. This command affects only the current buffer.

12.5 Optional Display Features

To add the current line number of point to the mode line, enable Line Number mode with the command `M-x line-number-mode`. The line number appears before *pos*, with the letter ‘`L`’ to indicate what it is. See Section 31.1 [Minor Modes], page 291, 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.

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* `[`*d*`]`

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. *d* is an approximate index of the ratio of disk activity to cpu activity for all users. (Some fields may be missing if your operating system cannot support them.)

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

Customization note: the variable `mode-line-inverse-video` controls whether the mode line is displayed in inverse video (assuming the terminal supports it); `nil` means no inverse video. The default is `t`. See Section 12.6 [Display Vars], page 73.

12.6 Variables Controlling Display

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

The variable `mode-line-inverse-video` controls whether the mode line is displayed in inverse video (assuming the terminal supports it); `nil` means don't do so. See Chapter 2 [Mode Line], page 19.

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 1.2 [Echo Area], page 16.

If the variable `ctl-arrow` is `nil`, control characters in the buffer are displayed with octal escape sequences, all except newline and tab. Altering the value of `ctl-arrow` makes it local to the current buffer; until that time, the default value is in effect. The default is initially `t`. See section "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 you set the variable `selective-display-ellipses` to `nil`, the three dots do not appear at the end of a line that precedes invisible lines. Then there is no visible indication of the invisible lines. This variable too becomes local automatically when set.

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 4.7 [Continuation Lines], page 29. 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 17.2 [Split Window], page 126. See also section "Display" in *The Emacs Lisp Reference Manual*.

The variable `baud-rate` holds the 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.)

13 Searching and Replacement

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

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

13.1 Incremental Search

An incremental search begins searching as soon as you type the first character of the search string. As you type in the search string, Emacs shows you where the string (as you have typed it so far) would be found. When you have typed enough characters to identify the place you want, you can stop. Depending on what you 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. Meanwhile, the search string '`FOO`' has been echoed 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 looking for. In this event, type another C-s to move to the next occurrence of the search string. This can be done any number of times. If you overshoot, you can cancel some C-s characters with DEL.

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

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 13.6 [Search Case], page 84.

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

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

To search for a newline, type LFD (also known as 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 meaning as an Emacs command: it causes the following character to be treated the way a graphic character would normally be treated in the same context. You can also specify a character by its octal code: enter `C-q` followed by three 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 cancelled with `DEL`.

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

The characters `C-y` and `C-w` can be used in incremental search to grab text from the buffer into the search string. This makes it convenient to search for another occurrence of text at point. `C-w` copies the word after point as part of the search string, advancing point over that word. Another `C-s` to repeat the search will then search for a string including that word. `C-y` is similar to `C-w` but copies all the rest of the current line into the search string. Both `C-y` and `C-w` convert the text they copy to lower case if the search is current not case-sensitive; this is so the search remains case-insensitive.

The character `M-y` copies the most recent killed text into the search string.

To customize the special characters that incremental search understands, alter their bindings in the keymap `isearch-mode-map`.

13.1.1 Slow Terminal Incremental Search

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

When 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`. 1 is its normal value.

13.2 Nonincremental Search

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

`C-s 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.

13.3 Word Search

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

Word search is useful 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 13.4 [Regexp Search], page 79.

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.

13.4 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. To search backward in the buffer, use C-M-r (isearch-backward-regexp).

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.

13.5 Syntax of Regular Expressions

Regular expressions have a syntax in which a few characters are special constructs and the rest are *ordinary*. An ordinary character is a simple regular expression which matches that 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 which begins with 'a' and ends with 'b'.

*
: is not a construct by itself; it is a postfix operator, which 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, 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 character, 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 character, 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 a '`]`'. 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 a character set, by writing two characters with a '`-`' between them. Thus, '`[a-z]`' matches any lower-case letter. Ranges may be intermixed freely with individual characters, as in '`[a-z$%.]`', which matches any lower case letter or '`$`', '`%`' or period.

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

To include a '`]`' in a character set, you must make it the first character. For example, '`[]a]`' matches '`]`' or '`a`'. To include a '`-`', write '`-`' at the beginning or end of a range. To include '`^`', make it other than the first character in the set.

`[^ ... ]` '`[^`' 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 ('`-`' 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`' which occurs at the beginning of a line.

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

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

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

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

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

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

  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 scope 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 mark a matched substring for future reference.

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

`\`*d*
: 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 order that the open-parentheses appear in the regular expression. '\1' through '\9' refer to the text previously matched by the corresponding '\(. . . \)' construct.

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

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, provided it is at the beginning of the buffer.

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

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

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

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

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

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

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

\s*c* matches any character whose syntax is *c*. Here *c* is a character which represents a syntax code: thus, 'w' for word constituent, '(' 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 31.6 [Syntax], page 311).

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-q C-j` to enter a newline. You would also type single slashes as themselves, instead of doubling them for Lisp syntax.

13.6 Searching and Case

All sorts of searches in Emacs normally ignore the case of the text they are searching through, if you specify 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 in the search string makes the search case-sensitive. Thus, searching for '`Foo`' does not find '`foo`' or '`FOO`'. This applies to regular expression search also. If you delete the upper-case character from the search string, it ceases to have this effect.

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 31.2.4 [Locals], page 295.

13.7 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 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 24.2 [Expanding Abbrevs], page 212.

---

[1] In some editors, search-and-replace operations are the only convenient way to make a single change in the text.

13.7.1 Unconditional Replacement

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

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

To replace every instance of '`foo`' after point with '`bar`', use the command `M-x replace-string` with the two arguments '`foo`' and '`bar`'. Replacement 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 30.4 [Narrowing], page 282).

When `replace-string` exits, point is left at the last occurrence replaced. The position of point where the `replace-string` command was issued is remembered on the mark ring; use `C-u C-SPC` to move back there.

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

13.7.2 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 text 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 give '`\\`'. For example,

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

13.7.3 Replace Commands and Case

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

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

replaces a lower case '`foo`' with a lower case '`bar`', '`FOO`' with '`BAR`', and '`Foo`' with '`Bar`'. If upper case letters are used in the second argument, they remain upper case every time that argument is inserted. If upper case letters are used in the first argument, the second argument is always substituted exactly as given, with no case conversion. Likewise, if the variable `case-replace` is set to `nil`, replacement is done without case conversion. If `case-fold-search` is set to `nil`, case is significant in matching occurrences of '`foo`' to replace; this also inhibits case conversion of the replacement string.

13.7.4 Query Replace

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

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

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

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, except that since the replacement has already been made, `DEL` and `SPC` are equivalent. You could type `C-r` at this point (see below) to alter the replaced text. You could also type `C-x u` to undo the replacement; this exits the `query-replace`, so if you want to do further replacement you must use `C-x ESC RET` to restart (see Section 6.5 [Repetition], page 40).

`RET`
`ESC`
: 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 allowed, 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 30.9 [Recursive Edit], page 287.

`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 `ESC`.

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`, which repeats the `query-replace` because it used the minibuffer to read its arguments. See Section 6.5 [Repetition], page 40.

See also Section 28.8 [Transforming File Names], page 247, for Dired commands to rename, copy, or link files by replacing regexp matches in file names.

13.8 Other Search-and-Loop Commands

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

`M-x occur RET` *regexp* `RET`
: Print each line that follows point and 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.

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

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

`M-x count-matches 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*.

14 Commands for Fixing Typos

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

14.1 Killing Your Mistakes

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

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

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

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

When your mistake is longer than a couple of characters, it might be more convenient to use `M-DEL` or `C-x DEL`. `M-DEL` kills back to the start of the last word, and `C-x DEL` kills back to the start of the last sentence. `C-x DEL` is particularly useful when you 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 10.2 [Yanking], page 59.

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

14.2 Transposing Text

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

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

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

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

The common error of transposing two characters can be fixed, when they are adjacent, with the `C-t` command (`transpose-chars`). Normally, `C-t` transposes the two characters on either side of point. When given at the end of a line, rather than transposing the last character of the line with the newline, which would be useless, `C-t` transposes the last two characters on the line. So, if you catch your transposition error right away, you can fix it with just a `C-t`. If you don't catch it so fast, you must move the cursor back to between the two transposed characters. If you transposed a space with the last character of the word before it, the word motion commands are a good way of getting there. Otherwise, a reverse search (`C-r`) is often the best way. See Chapter 13 [Search], page 75.

`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 22.2 [Lists], page 164), 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.

14.3 Case Conversion

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

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

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

A very common error is to type words in the wrong case. Because of this, the word case-conversion commands `M-l`, `M-u` and `M-c` have a special feature when used with a negative argument: they do not move the cursor. As soon as you see you have mistyped the last word, you can simply case-convert it and go on typing. See Section 21.6 [Case], page 151.

14.4 Checking and Correcting Spelling

This section describes the commands to check the spelling of a single word or of a portion of a buffer.

`M-$` Check and correct spelling of word at point (`ispell-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-string RET` *word* `RET`
: Check spelling of *word*.

`M-x ispell-complete-word`
: Complete the word before point based on the spelling dictionary.

`M-x reload-ispell`
: Make the Ispell subprocess reread your private dictionary.

`M-x kill-ispell`
: Kill the Ispell subprocess.

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. Each time these commands encounter an incorrect word, they ask you what to do.

Whenever one of these commands finds an incorrect word, it displays a list of alternatives, usually including several "near-misses"—words that are close to the word being checked. 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*.

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.

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

`l` *regexp* `RET`
: Look in the dictionary for words that match *regexp*. These words become the new list of "near-misses"; you can select one of them to replace with by typing a digit.

`C-g` Quit interactive spell checking. You can restart it again afterward with `C-u M-$`.

The command `ispell-complete-word`, which is bound to the key `M-TAB` in Text mode and related modes, performs completion based on spelling correction. Insert the beginning of a word, and then type `M-TAB`; the command inserts as many more letters as can be uniquely determined from the letters in the buffer, based on your currently loaded dictionaries. See Section 21.7 [Text Mode], page 152.

The first time you use any of the spell checking commands, it starts an Ispell subprocess. The first thing the subprocess does is read your private dictionary, which is the file '`~/ispell.words`'. Words that you "insert" with the `i` command are added to that file, but not right away—only at the end of the interactive replacement procedure. Use the `M-x reload-ispell` command to reload your private dictionary from '`~/ispell.words`' if you edit the file outside of Ispell.

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 kill-ispell`. This is not usually necessary, since the process uses no time except when you do spelling correction.

15 File Handling

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

In addition to visiting and saving files, Emacs can delete, copy, rename, and append to files, keep multiple versions of them, and operate on file directories.

15.1 File Names

Most Emacs commands that operate on a file require you to specify the file name. (Saving and reverting are exceptions; the buffer knows which file name to use for them.) You enter the file name using the minibuffer (see Chapter 6 [Minibuffer], page 35). *Completion* is available, to make it easier to specify long file names. See Section 6.3 [Completion], page 37.

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`', the double slash says to ignore the default directory and use just the name that you typed. See Section 6.1 [Minibuffer File], page 35.

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*.

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

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

15.2 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 C-f` Visit a file, in another window (`find-file-other-window`). Don't change the selected window.

`C-x 5 C-f` Visit a file, in a new frame (`find-file-other-frame`). Don't change the selected frame.

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 Emacs are made in the Emacs buffer. They do not take effect in the file that you visited, or any place permanent, until you *save* the buffer. Saving the buffer means that Emacs writes the current contents of the buffer into its visited file. See Section 15.3 [Saving], page 96.

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 Chapter 6 [Minibuffer], page 35), with defaulting and completion in the standard manner (see Section 15.1 [File Names], page 93). 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 15.3.2 [Simultaneous Editing], page 101.

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, Dired is called on that directory (see Chapter 28 [Dired], page 241). This can be inhibited by setting the variable `find-file-run-dired` to `nil`; then it is an error to try to visit a directory.

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` (`toggle-read-only`). See Section 16.3 [Misc Buffer], page 120.

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.

`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 Chapter 17 [Windows], page 125.

`C-x 5 f` (`find-file-other-frame`) is similar, but opens a new frame. This feature is available only when you are using a window system. See Chapter 18 [Frames], page 129.

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.

There are several ways to specify automatically the major mode for editing the file (see Section 19.1 [Choosing Modes], page 137), and to specify local variables defined for that file (see Section 31.2.5 [File Variables], page 297).

15.3 Saving Files

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

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

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

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

`C-x C-w` Save the current buffer in a specified file (`write-file`).

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

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

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

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

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

The command `C-x s` (`save-some-buffers`) offers to save any or all modified buffers. It asks you what to do with each buffer. The options 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.

`ESC` 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 and do not want the changes to be saved, 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 it by mistake. One thing you can do is type `M-~` (`not-modified`), which clears out the indication that the buffer is modified. If you do this, none of the save commands will believe that the buffer needs to be saved. ('~' is often used as a mathematical symbol for 'not'; thus 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 15.4 [Reverting], page 103. 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 with a file has the same effect as C-x C-w; that is, it reads a file name, marks the buffer as visiting that file, and saves it there. The default file name in a buffer that is not visiting a file is made by combining the buffer name with the buffer's default directory.

If Emacs is about to save a file and sees that the date of the latest version on disk does not match what Emacs last read or wrote, Emacs notifies you of this fact, because it probably indicates a problem caused by simultaneous editing and requires your immediate attention. See Section 15.3.2 [Simultaneous Editing], page 101.

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

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

The list local-write-file-hooks is used just like write-file-hooks. This list is for use as a local variable. It is a permanent local, so that switching major modes does not affect it.

The variable write-contents-hooks holds another list of functions to be called before writing out a buffer to a file. These functions are used just like the ones in `write-file-hooks`. The difference between this and `write-file-hooks` is that changing the major mode does clear this variable, and setting the visited file name does not clear it.

The hook functions are responsible for writing backup files, if you want that to be done. Here is how:

```
(or buffer-backed-up (backup-buffer))
```

You might also wish to save the file modes value returned by `backup-buffer` and use that to set the mode bits of the file that you write. This is what happens ordinarily when backups are made.

The hook variables described here are not normal hooks, because the values returned by the hook functions do matter. This is why they have names ending in '`-hooks`' rather than '`-hook`'. See Section 31.2.3 [Hooks], page 294.

15.3.1 Backup Files

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

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

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

15.3.1.1 Single or Numbered Backups

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

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

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

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

`t` Make numbered backups.

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

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

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 31.2.4 [Locals], page 295.

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`.

15.3.1.2 Automatic Deletion of Backups

To prevent unlimited consumption of disk space, Emacs can delete numbered backup versions automatically. Generally Emacs keeps the first few backups and the latest few backups, deleting any in between. This happens every time a new backup is made.

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

Dired's `.` (Period) command can also be used to delete old versions. See Section 28.3 [Dired Deletion], page 241.

15.3.1.3 Copying vs. Renaming

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

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

Having the owner change is usually a good idea, because then the owner always shows who last edited the file. Also, the owners of the backups show who produced those versions. Occasionally there is a file whose owner should not change; it is a good idea for such files to contain local variable lists to set `backup-by-copying-when-mismatch` locally (see Section 31.2.5 [File Variables], page 297).

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 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.

15.3.2 Protection against Simultaneous Editing

Simultaneous editing occurs when two users visit the same file, both make changes, and then both save them. If nobody were informed that this was happening, whichever user saved first would later find that his changes were lost. On some systems, Emacs notices immediately when the second user starts to change the file, and issues an immediate warning.

For the sake of systems where that is not possible, and in case someone else proceeds to change the file despite the warning, Emacs also checks when the file is saved, and issues a second warning if you are about to overwrite a file containing another user's changes. You can prevent loss of the other user's work by taking the proper corrective action at that time.

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 writing another file in a directory reserved for this purpose.) The lock is removed when you save the changes. The idea is that the file is locked whenever 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. On these systems, 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.

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 15.8 [Directory Listing], page 115). 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.

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

15.4 Reverting a Buffer

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

`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 randomly with `C-x b` cannot be reverted; `revert-buffer` reports an error when asked to do so.

15.5 Auto-Saving: Protection Against Disasters

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

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

15.5.1 Auto-Save Files

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

Normally, the auto-save file name is made by appending '`#`' to the front and rear of the visited file name. Thus, a buffer visiting file '`foo.c`' 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.

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.

When you delete a large amount of a buffer's text, auto-saving turns off 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 restart auto-saving in that buffer, use M-x auto-save with a positive argument.

15.5.2 Controlling Auto-Saving

Each time you visit a file, auto-saving is turned on for that file's buffer if the variable auto-save-default is non-nil (but not in batch mode; see Chapter 3 [Entering Emacs], page 23). 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`.

15.5.3 Recovering Data from Auto-Saves

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

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

15.6 File Name Aliases

Symbolic links and hard links both make it possible for several file names to refer to the same file. For example, when '`foo`' is a symbolic link to '`bar`', the file has two names, but '`bar`' is the real name, and '`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`.

15.7 Version Control

Version control systems are packages that can record multiple versions of a source file, usually storing the unchanged parts of the file just once. Version control systems also record history information such as the creation time of each version, who created it, and a description of what was changed in that version.

The GNU project recommends the version control system known as RCS, which is free software and available from the Free Software Foundation. Emacs supports use of either RCS or SCCS (a proprietary, but widely used, version control system that is not quite as powerful as RCS) through a facility called VC. The same Emacs commands work with either RCS or SCCS, so you hardly have to know which one of them you are using.

15.7.1 Concepts of Version Control

When a file is under version control, we also say that it is *registered* in the version control system. Each registered file has a corresponding *master file* which represents the file's present state plus its change history, so that you can reconstruct from it either the current version or any specified earlier version. Usually the master file also records a *log entry* for each version describing what was changed in that version.

The file that is maintained under version control is sometimes called the *work file* corresponding to its master file.

To examine a file, you *check it out*. This extracts a version of the source file (typically, the most recent) from the master file. If you want to edit the file, you must check it out *locked*. Only one user can do this at a time for any given source file. (This kind of locking is completely unrelated to the locking that Emacs uses to detect simultaneous editing of a file.)

When you are done with your editing, you must *check in* the new version. This records the new version in the master file, and unlocks the source file so that other people can lock it and thus modify it.

Checkin and checkout are the basic operations of version control. You can do both of them with a single Emacs command: `C-x C-q` (`vc-toggle-read-only`).

15.7.2 Editing with Version Control

When you visit a file that is maintained using version control, the mode line displays '`RCS`' or '`SCCS`' to inform you that version control is in use, and also (in case you care) which low-level system the file is actually stored in. Normally, such a source file is read-only, and the mode line indicates this with '%%'. With RCS, the mode line also indicates the number of the head version, which is normally also the version you are looking at.

These are the commands for editing a file maintained with version control:

`C-x C-q` Check the visited file in or out.

`C-x v u` Revert the buffer and the file to the last checked in version.

`C-x v c` Remove the last-entered change from the master for the visited file. This undoes your last check-in.

`C-x v i` Register the visited file in version control.

(`C-x v` is the prefix key for version control commands; all of these commands except for `C-x C-q` start with `C-x v`.)

When you want to modify a file maintained with version control, type `C-x C-q` (`vc-toggle-read-only`). This *checks out* the file, and tells RCS or SCCS to lock the file. This means making the file writable for you (but not for anyone else).

When you are finished editing the file, type `C-x C-q` again. When used on a file that is checked out, this command checks the file in. But check-in does not start immediately; first, you must enter the *log entry*—a description of the changes in the new version. `C-x C-q` pops up a buffer for you to enter this in. When you are finished typing in the log entry, type `C-c C-c` to terminate it; this is when actual check-in takes place.

Once you have checked in your changes, the file is unlocked, so that other users can lock it and modify it.

Emacs does not save backup files for source files that are maintained with version control. If you want to make backup files despite version control, set the variable `vc-make-backup-files` to a non-`nil` value.

Normally the work file exists all the time, whether it is locked or not. If you set `vc-keep-workfiles` to `nil`, then checking in a new version with `C-x C-q` deletes the work file; but any attempt to visit the file with Emacs creates it again.

It is not impossible to lock a file that someone else has locked. If you try to check out a file that is locked, `C-x C-q` asks you whether you want to "steal the lock." If you say yes, the file becomes locked by you, but a message is sent to the person who had formerly locked the file, to inform him of what has happened. The mode line indicates that a file is locked by someone else by displaying the login name of that person, before the version number.

If you want to discard your current set of changes and revert to the last version checked in, use `C-x v u` (`vc-revert-buffer`). This cancels your last check-out, leaving the file unlocked. If you want to make a different set of changes, you must first check the file out again. `C-x v u` requires confirmation, unless it sees that you haven't made any changes since the last checked-in version.

`C-x v u` is also the command to use if you lock a file and then don't actually change it.

You can cancel a change after checking it in, with `C-x v c` (`vc-cancel-version`). This command discards all record of the most recent checked in version, so be careful about using it. It requires confirmation with `yes`. By default, `C-x v c` reverts your workfile and buffer to the previous version (the one that precedes the version that is deleted), but you can prevent the reversion by giving the command a prefix argument. Then the buffer does not change.

This command with a prefix argument is useful when you have checked in a change and then discover a trivial error in it; you can cancel the erroneous check-in, fix the error, and repeat the check-in.

Be careful when invoking `C-x v c`, as it is easy to throw away a lot of work with it. To help you be careful, this command always asks for confirmation with '`yes`'.

You can register the visited file for version control using `C-x v i` (`vc-register`). If the variable `vc-default-back-end` is non-`nil`, it specifies which version control system to use; otherwise, this uses RCS if it is installed on your system and SCCS if not. After `C-x v i`, the file is unlocked and read-only. Type `C-x C-q` if you wish to edit it.

By default, the initial version number is 1.1. If you want to use a different number, give `C-x v i` a prefix argument; then it reads the initial version number using the minibuffer.

If `vc-initial-comment` is non-`nil`, `C-x v i` reads an initial comment (much like a log entry) to describe the purpose of this source file.

To specify the version number for a subsequent checkin, use the command `C-u C-x v v`. `C-x v v` (`vc-next-action`) is the command that `C-x C-q` uses to do the "real work" when the visited file uses version control. When used for checkin, and given a prefix argument, it reads the version number with the minibuffer.

15.7.3 Variables Affecting Check-in and Check-out

If `vc-suppress-confirm` is non-`nil`, then `C-x C-q` and `C-x v i` can save the current buffer without asking, and `C-x v u` also operates without asking for confirmation. (This variable does not affect `C-x v c`; that is so drastic that it should always ask for confirmation.)

VC mode does much of its work by running the shell commands for RCS and SCCS. If `vc-command-messages` is non-`nil`, VC displays messages to indicate which shell commands it runs, and additional messages when the commands finish.

Normally, VC assumes that it can deduce the locked/unlocked state of files by looking at the file permissions of the work file; this is fast. However, if the 'RCS' or 'SCCS' subdirectory is actually a symbolic link, then VC does not trust the file permissions to reflect this status.

You can specify the criterion for whether to trust the file permissions by setting the variable `vc-mistrust-permissions`. Its value may be `t` (always mistrust the file permissions and check the master file), `nil` (always trust the file permissions), or a function of one argument which makes the decision. The argument is the directory name of the 'RCS' or 'SCCS' subdirectory. A non-`nil` value from the function says to mistrust the file permissions.

If you find that the file permissions of work files are changed erroneously, set `vc-mistrust-permissions` to `t`. Then VC always checks the master file to determine the file's status.

You can specify additional directories to search for version control programs by setting the variable `vc-path`. These directories are searched before the usual search path. The proper result usually happens automatically.

15.7.4 Log Entries

When you're editing an initial comment or log entry for inclusion in a master file, finish your entry by typing `C-c C-c`.

`C-c C-c` Finish the comment edit normally (`vc-finish-logentry`). This finishes check-in.

To abort check-in, just don't type `C-c C-c` in that buffer. You can switch buffers and do other editing. As long as you don't try to check in another file, the entry you were editing remains in its buffer, and you can go back to that buffer at any time to complete the check-in.

If you change several source files for the same reason, it is often convenient to specify the same log entry for many of the files. To do this, use the history of previous log entries. The commands `M-n`, `M-p`, `M-s` and `M-r` for doing this work just like the minibuffer history commands (except that these versions are used outside the minibuffer).

Each time you check in a file, the log entry buffer is put into VC Log mode, which involves running two hooks: `text-mode-hook` and `vc-log-mode-hook`. See Section 31.2.3 [Hooks], page 294.

15.7.5 Change Logs and VC

If you use RCS for a program and also maintain a change log file for it (see Section 22.11 [Change Log], page 178), you can generate change log entries automatically from the version control log entries:

`C-x v a` Visit the current directory's change log file and create new entries for versions checked in since the most recent entry in the change log file (`vc-update-change-log`).

This command works with RCS only; it does not work with SCCS.

For example, suppose the first line of '`ChangeLog`' is dated 10 April 1992, and that the only check-in since then was by Nathaniel Bowditch to '`rcs2log`' on 8 May 1992 with log text '`Ignore log messages that start with '#'.`'. Then `C-x v a` visits '`ChangeLog`' and inserts text like this:

```
Fri May  8 21:45:00 1992  Nathaniel Bowditch  (nat@apn.org)

        * rcs2log: Ignore log messages that start with '#'.
```

You can then edit the new change log entry further as you wish.

Normally, the log entry for file '`foo`' is displayed as '`* foo:` *text of log entry*'. The '`:`' after '`foo`' is omitted if the text of the log entry starts with '`(`*functionname*`): `'. For example, if the log entry for '`vc.el`' is '`(vc-do-command): Check call-process status.`', then the text in '`ChangeLog`' looks like this:

```
Wed May  6 10:53:00 1992  Nathaniel Bowditch  (nat@apn.org)

        * vc.el (vc-do-command): Check call-process status.
```

When `C-x v a` adds several change log entries at once, it groups related log entries together if they all are checked in by the same author at nearly the same time. If the log entries for several such files all have the same text, it coalesces them into a single entry. For example, suppose the most recent checkins have the following log entries:

For '`vc.texinfo`':
    `Fix expansion typos.`
For '`vc.el`':
    `Don't call expand-file-name.`
For '`vc-hooks.el`':
    `Don't call expand-file-name.`

They appear like this in '`ChangeLog`':

```
Wed Apr  1 08:57:59 1992  Nathaniel Bowditch  (nat@apn.org)

        * vc.texinfo: Fix expansion typos.

        * vc.el, vc-hooks.el: Don't call expand-file-name.
```

Normally, `C-x v a` separates log entries by a blank line, but you can mark several related log entries to be clumped together (without an intervening blank line) by starting the text of each related log entry with a label of the form '`{`*clumpname*`}` '. The label itself is not copied to '`ChangeLog`'. For example, suppose the log entries are:

For '`vc.texinfo`':
    `{expand} Fix expansion typos.`
For '`vc.el`':
    `{expand} Don't call expand-file-name.`
For '`vc-hooks.el`':
    `{expand} Don't call expand-file-name.`

Then the text in '`ChangeLog`' looks like this:

```
Wed Apr  1 08:57:59 1992  Nathaniel Bowditch  (nat@apn.org)

        * vc.texinfo: Fix expansion typos.
        * vc.el, vc-hooks.el: Don't call expand-file-name.
```

A log entry whose text begins with '`#`' is not copied to '`ChangeLog`'. For example, if you merely fix some misspellings in comments, you can log the change with an entry beginning with '`#`' to avoid putting such trivia into '`ChangeLog`'.

15.7.6 Examining And Comparing Old Versions

`C-x v ~` *version* `RET`
: Examine version *version* of the visited file, in a buffer of its own (`vc-version-other-window`).

`C-x v =`
: Compare the current buffer contents with the latest checked-in version of the file.

`C-u C-x v =` *file* `RET` *oldvers* `RET` *newvers* `RET`
: Compare the specified two versions of *file*.

You can examine any version of a file by first visiting it, and then using `C-x v ~` *version* `RET` (`vc-version-other-window`). This puts the text of version *version* in a file named '*filename*`.~`*version*`~`', then visits it in a separate window.

To compare two versions of a file, use the command `C-x v =` (`vc-diff`).

Plain `C-x v =` compares the current buffer contents (saving them in the file if necessary) with the last checked-in version of the file. With a prefix argument, `C-x v =` reads a file name and two version numbers, then compares those versions of the specified file.

If you supply a directory name instead of the name of a work file, this command compares the two specified versions of all registered files in that directory and its subdirectories. You can also specify a snapshot name (see Section 15.7.9 [Snapshots], page 113) instead of one or both version numbers.

You can specify a checked-in version by its number; you can specify the most recent checked-in version with an empty version number.

This command works by running the `diff` utility, getting the options from the variable `diff-switches`. It displays the output in a special buffer in another window. Unlike the `M-x diff` command, `C-x v =` does not try to find the changes in the old and new versions. This is because one or both versions normally do not exist as files. They exist only in the records of the master file. See Section 15.9 [Comparing Files], page 116, for more information about `M-x diff`.

15.7.7 VC Status Commands

To view the detailed version control status and history of a file, type `C-x v l` (`vc-print-log`). It displays the history of changes to the current file, including the text of the log entries. The output appears in a separate window.

When you are working on a large program, it's often useful to find all the files that are currently locked, or all the files maintained in version control at all. You can use `C-x v d` (`vc-directory`) to show all the locked files in or beneath the current directory. This includes all files that are locked by any user. `C-u C-x v d` lists all files in or beneath the current directory that are maintained with version control.

The list of files is displayed as a buffer that uses an augmented Dired mode. The names of the users locking various files are shown (in parentheses) in place of the owner and group. All the normal Dired commands work in this buffer. Most interactive VC commands work also, and apply to the file name on the current line.

The `C-x v v` command (`vc-next-action`), when used in the augmented Dired buffer, operates on all the marked files (or the file on the current line). If it operates on more than one file, it handles each file according to its current state; thus, it may check out one file and check in another (because it is already checked out). If it has to check in any files, it reads a single log entry, then uses that text for all the files being checked in. This can be convenient for registering or checking in several files at once, as part of the same change.

15.7.8 Renaming VC Work Files and Master Files

When you rename a registered file, you must also rename its master file correspondingly to get proper results. Use `vc-rename-file` to rename the source file as you specify, and rename its master file accordingly. It also updates any snapshots (see Section 15.7.9 [Snapshots], page 113) that mention the file, so that they use the new name; despite this, the snapshot thus modified may not completely work (see Section 15.7.9.2 [Snapshot Caveats], page 114).

You cannot use `vc-rename-file` on a file that is locked by someone else.

15.7.9 Snapshots

A *snapshot* is a named set of file versions (one for each registered file) that you can treat as a unit. One important kind of snapshot is a *release*, a (theoretically) stable version of the system that is ready for distribution to users.

15.7.9.1 Making and Using Snapshots

There are two basic commands for snapshots; one makes a snapshot with a given name, the other retrieves a named snapshot.

`C-x v s` *name* `RET`
: Define the last saved versions of every registered file in or under the current directory as a snapshot named *name* (`vc-create-snapshot`).

`C-x v r` *name* `RET`
: Check out all registered files at or below the current directory level using whatever versions correspond to the snapshot *name* (`vc-retrieve-snapshot`).

: This command reports an error if any files are locked at or below the current directory, without changing anything; this is to avoid overwriting work in progress.

A snapshot uses a very small amount of resources—just enough to record the list of file names and which version belongs to the snapshot. Thus, you need not hesitate to create snapshots whenever they are useful.

You can give a snapshot name as an argument to `C-x v =` or `C-x v ~` (see Section 15.7.6 [Old Versions], page 111). Thus, you can use it to compare a snapshot against the current files, or two snapshots against each other, or a snapshot against a named version.

15.7.9.2 Snapshot Caveats

VC's snapshot facilities are modeled on RCS's named-configuration support. They use RCS's native facilities for this, so under VC snapshots made using RCS are visible even when you bypass VC.

For SCCS, VC implements snapshots itself. The files it uses contain name/file/version-number triples. These snapshots are visible only through VC.

File renaming and deletion can create some difficulties with snapshots. This is not a VC-specific problem, but a general design issue in version control systems that no one has solved very well yet.

If you rename a registered file, you need to rename its master along with it (the command `vc-rename-file` does this automatically). If you are using SCCS, you must also update the records of the snapshot, to mention the file by its new name (`vc-rename-file` does this, too). An old snapshot that refers to a master file that no longer exists under the recorded name is invalid; VC can no longer retrieve it. It would be beyond the scope of this manual to explain enough about RCS and SCCS to explain how to update the snapshots by hand.

Using `vc-rename-file` makes the snapshot remain valid for retrieval, but it does not solve all problems. For example, some of the files in the program probably refer to others by name. At the very least, the makefile probably mentions the file that you renamed. If you retrieve an old snapshot, the renamed file is retrieved under its new name, which is not the name that the makefile expects. So the program won't really work as retrieved.

15.7.10 Inserting Version Control Headers

Sometimes it is convenient to put version identification strings directly into working files. Certain special strings called *version headers* are replaced in each successive version by the number of that version.

You can use the `C-x v h` command (`vc-insert-headers`) to insert a suitable header string.

`C-x v h` Insert headers in a file for use with your version-control system.

The default header string is '`$Id$`' for RCS and '`%W%`' for SCCS. You can specify other headers to insert by setting the variable `vc-header-alist`. Its value is a list of elements of the form `(`*program*` . `*string*`)` where *program* is `RCS` or `SCCS` and *string* is the string to use.

Instead of a single string, you can specify a list of strings; then each string in the list is inserted as a separate header on a line of its own.

It is often necessary to use "superfluous" backslashes when writing the strings that you put in this variable. This is to prevent the string in the constant from being interpreted as a header itself if the Emacs Lisp file containing it is maintained with version control.

Each header is inserted surrounded by tabs, inside comment delimiters, on a new line at the start of the buffer. Normally the ordinary comment start and comment end strings of the current mode are used, but for certain modes, there are special comment delimiters for this purpose; the variable `vc-comment-alist` specifies them. Each element of this list has the form `(`*mode starter ender*`)`.

The variable `vc-static-header-alist` specifies further strings to add based on the name of the buffer. Its value should be a list of elements of the form `(`*regexp* `.` *format*`)`. Whenever *regexp* matches the buffer name, *format* is inserted as part of the header. A header line is inserted for each element that matches the buffer name, and for each string specified by `vc-header-alist`. The header line is made by processing the string from `vc-header-alist` with the format taken from the element. The default value for `vc-static-header-alist` is:

```
(("\\.c$" .
  "\n#ifndef lint\nstatic char vcid[] = \"\%s\";\n\
#endif /* lint */\n"))
```

which specifies insertion of a string of this form:

```

#ifndef lint
static char vcid[] = "string";
#endif /* lint */
```

15.8 Listing a File Directory

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

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

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

The command to 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`'. An example of specifying a file name pattern is

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

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

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

15.9 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 `C-c C-c` to find 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 23.1 [Compilation], page 199.

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. Point moves forward in each window, a character at a time in each window, until the next characters in the two windows are different. Then the command is finished. For more information about windows in Emacs, Chapter 17 [Windows], page 125.

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 22.13 [Emerge], page 184, for convenient facilities for merging two similar files.

15.10 Miscellaneous File Operations

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

`M-x view-file` allows you to scan or read a file by sequential screenfuls. It reads a file name argument using the minibuffer. After reading the file into an Emacs buffer, `view-file` 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 `C-h` while viewing for a list of them. They are mostly the same as normal Emacs cursor motion commands. To exit from viewing, type `C-c`. 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 16.3 [Misc Buffer], page 120.

`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 10.3 [Accumulating Text], page 62.

`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 Chapter 28 [Dired], page 241).

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

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

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

`M-x make-symbolic-link` reads two file names *old* and *linkname*, then creates a symbolic link named *linkname* and pointing at *old*. The effect is that future attempts to open file *linkname* will refer to whatever file is named *old* at the time the opening is done, or will get an error if the name *old* is not in use at that time. This command does not expand the argument *filename*, 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.

16 Using Multiple Buffers

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

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

When Emacs 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 Chapter 17 [Windows], page 125).

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

Each buffer records individually what file it is visiting, whether it is modified, and what major mode and minor modes are in effect in it (see Chapter 19 [Major Modes], page 137). 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 31.2.4 [Locals], page 295.

16.1 Creating and Selecting Buffers

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

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

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 6.3 [Completion], page 37). An empty argument to `C-x b` specifies the most recently selected buffer that is not displayed in any window.

Most buffers are created by visiting files, or by Emacs commands that want to display some text, but you can also create a buffer explicitly by typing `C-x b` *bufname* `RET`. This makes a new, empty buffer which is not visiting any file, and selects it for editing. Such buffers are used for making notes to yourself. If you try to save one, you are asked for the file name to use. The new buffer's major mode is determined by the value of `default-major-mode` (see Chapter 19 [Major Modes], page 137).

Note that `C-x C-f`, and any other command for visiting a file, can also be used to switch buffers. See Section 15.2 [Visiting], page 94.

16.2 Listing Existing Buffers

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

To print a list of all the buffers that exist, type `C-x C-b`. Each line in the list shows one buffer's name, major mode and visited file. The buffers are listed in the order, most recently visited 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 15.3 [Saving], page 96). '`%`' 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/`'.

16.3 Miscellaneous Buffer Operations

`C-x C-q` Toggle read-only status of buffer (`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 '`%%`' near the left margin.

Read-only buffers are made by subsystems such as Dired and Rmail that have special commands to operate on the text; also if you visit a file that is protected so you cannot write it. If you wish to make changes in a read-only buffer, use the command `C-x C-q` (`vc-toggle-read-only`). It makes a read-only buffer writable, and makes a writable buffer read-only. This works by setting the variable `buffer-read-only`, which has a local value in each buffer and makes the buffer read-only if its value is non-`nil`.

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 buffer.

`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*`', then do `M-x shell` again, it makes a new shell buffer named '`*Shell*`'; meanwhile, the old shell buffer continues to exist under its altered name. This method is also good for mail buffers, compilation buffers, and any Emacs feature which creates a special buffer with a particular name.

`M-x view-buffer` is much like `M-x view-file` (see Section 15.10 [Misc File Ops], page 116) except that it examines an already existing Emacs buffer. View mode provides commands for scrolling through the buffer conveniently but not for changing it. When you exit View mode, the value of point that resulted from your perusal remains in effect.

The commands `M-x append-to-buffer` and `M-x insert-buffer` can be used to copy text from one buffer to another. See Section 10.3 [Accumulating Text], page 62.

16.4 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. There are several 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 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`.

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 31.2.3 [Hooks], page 294).

16.5 Operating on Several Buffers

The *buffer-menu* facility is like a "Dired for buffers"; it allows you to request operations on various Emacs buffers by editing an Emacs buffer containing a list of them. You can save buffers, kill them (here called *deleting* them, for consistency with Dired), or display them.

`M-x buffer-menu`
: Begin editing a buffer listing all Emacs buffers.

The command `buffer-menu` writes a list of all Emacs buffers into the buffer '`*Buffer List*`', and selects that buffer in Buffer Menu mode. The buffer is read-only, and can 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`, `s` and `u` commands to add or remove flags also move down a 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 tag 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.

`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.

`m` Mark this line's buffer to be displayed in another window if the `q` command is used. The request shows as a '`>`' at the beginning of the line. The same buffer may not have both a delete request and a display request.

`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 select a suitable buffer, and turn on Buffer Menu mode. Everything else described above is implemented by the special commands provided in Buffer Menu mode. One consequence of this is that you can switch from the '`*Buffer List*`' buffer to another Emacs buffer, and edit there. You can reselect the `buffer-menu` buffer later, to perform the operations already requested, or you can kill it, or pay no further attention to it.

The only difference between `buffer-menu` and `list-buffers` is that `buffer-menu` selects the '`*Buffer List*`' buffer and `list-buffers` does not. If you run `list-buffers` (that is, type `C-x C-b`) and select the buffer list manually, you can use all of the commands described here.

The buffer '`*Buffer List*`' is not updated automatically; 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 repeat the `buffer-menu` command.

17 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 window belongs to one and only one frame.

17.1 Concepts of Emacs Windows

When multiple windows are being displayed, each window has an Emacs buffer designated for display in it. The same buffer may appear in more than one window; if it does, any changes in its text are displayed in all the windows where it appears. But the windows showing the same buffer can show different parts of it, because each window has its own value of point.

At any time, one of the windows is the *selected window*; the buffer this window is displaying is the current buffer. The terminal's cursor shows the location of point in this window. Each other window has a location of point as well, but since the terminal has only one cursor there is no way to show where those locations are. When you make multiple frames, each frame has a cursor which appears in the frame's selected window.

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. This means that in Transient Mark mode, each window highlights a different part of the buffer. The part that is highlighted in the selected window is the region that editing commands use.

Each window has its own mode line, which displays the buffer name, modification status and major and minor modes of the buffer that is displayed in the window. See Chapter 2 [Mode Line], page 19, for full details on the mode line.

17.2 Splitting Windows

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

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

The command `C-x 2` (`split-window-vertically`) breaks the selected window into two windows, one above the other. Both windows start out displaying the same buffer, with the same value of point. By default the two windows each get half the height of the window that was split; a numeric argument specifies how many lines to give to the top window.

`C-x 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; also, they do not always appear in inverse video, because the Emacs display routines have not been taught how to display a region of inverse video that is only part of a line on the screen.

When a window is less than the full width, text lines too long to fit are frequent. Continuing all those lines might be confusing. The variable `truncate-partial-width-windows` can be set non-`nil` to force truncation in all windows less than the full width of the screen, independent of the buffer being displayed and its value for `truncate-lines`. See Section 4.7 [Continuation Lines], page 29.

Horizontal scrolling is often used in side-by-side windows. See Chapter 12 [Display], page 69.

If `split-window-keep-point` is non-nil, `C-x 2` tries to avoid shifting any text on the screen by putting point in whichever window happens to contain the screen line the cursor is already on. The default is that `split-window-keep-point` is non-nil on slow terminals.

17.3 Using Other Windows

`C-x o` Select another window (`other-window`). That is `o`, not zero.

`C-M-v` Scroll the next window (`scroll-other-window`).

`M-x compare-windows`
Find next place where the text in the selected window does not match the text in the next window.

To select a different window, use `C-x o` (`other-window`). That is an `o`, for 'other', not a zero. When there are more than two windows, this command moves through all the windows in a cyclic order, generally top to bottom and left to right. 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 6.2 [Minibuffer Edit], page 36.

The usual scrolling commands (see Chapter 12 [Display], page 69) 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 15.9 [Comparing Files], page 116.

17.4 Displaying in Another Window

`C-x 4` is a prefix key for commands that select another window (splitting the window if there is only one) and select a buffer in that window. Different `C-x 4` commands have different ways of finding the buffer to select.

`C-x 4 b` *bufname* `RET`
: Select buffer *bufname* in another window. This runs `switch-to-buffer-other-window`.

`C-x 4 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 15.2 [Visiting], page 94.

`C-x 4 d` *directory* `RET`
: Select a Dired buffer for directory *directory* in another window. This runs `dired-other-window`. See Chapter 28 [Dired], page 241.

`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 Chapter 26 [Sending Mail], page 221).

`C-x 4 .` Find a tag in the current tag table in another window. This runs `find-tag-other-window`, the multiple-window variant of `M-.` (see Section 22.12 [Tags], page 179).

`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 15.2 [Visiting], page 94.

17.5 Deleting and Rearranging Windows

`C-x 0` Get rid of the selected window (`delete-window`). That is a zero.

`C-x 1` Get rid of all windows except the selected one (`delete-other-windows`).

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

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

To delete a window, type `C-x 0` (`delete-window`). (That is a zero.) The space occupied by the deleted window is 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 1` (`delete-other-windows`) is more powerful than `C-x 0`; it deletes all the windows except the selected one (and the minibuffer); the selected window expands to use the whole frame except for the echo area.

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. The extra screen space given to a window comes from one of its neighbors, if that is possible. 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`.

18 Frames and X Windows

When using the X Window System, you can create multiple windows at the X level in a single Emacs session. Each X window that belongs to Emacs displays a *frame* which can contain one or several Emacs windows. A frame initially contains a single general-purpose Emacs window which you can subdivide vertically or horizontally into smaller windows. A frame normally contains its own echo area and minibuffer, but you can make frames that don't have these—they use the echo area and minibuffer of another frame.

Anything you do in one frame also affects the other frames. For instance, if you put text in the kill ring in one frame, you can yank it in another frame. If you exit emacs through `C-x C-c` in one frame, it terminates all the frames. To delete just one frame, use `C-x 5 0`.

To avoid confusion, we reserve the word "window" for the subdivisions that Emacs implements, and never use it to refer to a frame.

18.1 Mouse Commands

`mouse-1`
: Move point to where you click (`mouse-set-point`). This is normally the left button.

`drag-mouse-1`
: Set the region to the text you select by dragging, and copy it to the kill ring (`mouse-set-region`). Thus, you can specify both ends of the region.

  If you move the mouse off the top or bottom of the window while dragging, the window scrolls at a steady rate until you move the mouse back into the window. This way, you can mark regions that don't fit entirely on the screen.

`mouse-2`
: Yank the last killed text, where you click (`mouse-yank-at-click`). This is normally the middle button.

`mouse-3`
: This command, `mouse-save-then-kill`, has several functions depending on where you click and the status of the region.

  If you have a highlighted region, or if the region was set just before by dragging button 1, `mouse-save-then-kill` adjusts the nearer end of the region by moving it to where you click. The adjusted region's text also replaces the old region's text in the kill ring.

  Otherwise, `mouse-save-then-kill` sets mark where you click, without changing point. It copies the new region to the kill ring.

  If you originally specified the region using a double or triple `MOUSE-1`, so that the region is defined to consist of entire words or lines, then adjusting the region also proceeds by entire words or lines.

If you use `mouse-save-then-kill` twice in a row at the same place, that kills the region already selected.

`double-mouse-1`
: This key sets the region around the word which you click on.

`double-drag-mouse-1`
: This key selects a region made up of the words that you drag across.

`triple-mouse-1`
: This key sets the region around the line which you click on.

`triple-drag-mouse-1`
: This key selects a region made up of the lines that you drag across.

The simplest way to kill some text is to press `MOUSE-1` at one end, then press `MOUSE-3` twice at the other end. See Section 10.1 [Killing], page 57. To copy the text into the kill ring without deleting it from the buffer, press `MOUSE-3` just once—or just drag across the text with `MOUSE-1`. Then you can copy it elsewhere by yanking it.

To yank the killed or copied text somewhere else, move the mouse there and press `MOUSE-2`. See Section 10.2 [Yanking], page 59.

The mouse commands for selecting a region are mostly compatible with the `xterm` program.

To copy text to another X window, kill it or save it in the kill ring. Under X, this also sets the *primary selection*. Then use the "paste" or "yank" command of the program operating the other window to insert the text from the selection.

To copy text from another X window, use the "cut" or "copy" command of the program operating the other window, to select the text you want. Then yank it in Emacs with `C-y` or `MOUSE-2`.

When Emacs puts text into the kill ring, or rotates text to the front of the kill ring, it sets the *primary selection* in the X server. This is how other X clients can access the text. Emacs also stores the text in the cut buffer, if the text is short enough (`x-cut-buffer-max` specifies the maximum number of characters); putting long strings in the cut buffer can be slow.

When you yank in Emacs, Emacs checks for a primary selection in another program; after that, it checks for text in the cut buffer.

18.2 Secondary Selection

The *secondary selection* is another way of selecting text using X. It does not use point or the mark, so you can use it to kill text without setting point or the mark.

`drag-M-mouse-1`
: Set the secondary selection, with one end at the place where you press down the button, and the other end at the place where you release it (`mouse-set-secondary`). In Transient Mark mode, highlighting appears and changes as you drag.

  If you move the mouse off the top or bottom of the window while dragging, the window scrolls at a steady rate until you move the mouse back into the window. This way, you can mark regions that don't fit entirely on the screen.

`M-mouse-1`
: Set one endpoint for the *secondary selection* (`mouse-start-secondary`).

`M-mouse-3`
: Make a secondary selection, using the place specified with `M-mouse-1` as the other end (`mouse-secondary-save-then-kill`). A second click at the same place kills the secondary selection just made.

`M-mouse-2`
: Kill the secondary selection (`mouse-kill-secondary`).

Double or triple clicking of `M-MOUSE-1` operates on words and lines, much like `MOUSE-1`.

18.3 Creating Frames

The prefix key `C-x 5` is analogous to `C-x 4`, with parallel subcommands. The difference is that `C-x 5` commands create a new frame rather than just a new window in the selected frame. (See Section 17.4 [Pop Up Window], page 127.) Different `C-x 5` commands have different ways of finding the buffer to select.

`C-x 5 b` *bufname* `RET`
: Select buffer *bufname* in another window. This runs `switch-to-buffer-other-frame`.

`C-x 5 f` *filename* `RET`
: Visit file *filename* and select its buffer in another frame. This runs `find-file-other-frame`. See Section 15.2 [Visiting], page 94.

`C-x 5 d` *directory* `RET`
: Select a Dired buffer for directory *directory* in another frame. This runs `dired-other-frame`. See Chapter 28 [Dired], page 241.

`C-x 5 m`
: Start composing a mail message in another frame. This runs `mail-other-frame`, and its same-frame version is `C-x m`. See Chapter 26 [Sending Mail], page 221.

`C-x 5 .`
: Find a tag in the current tag table in another frame. This runs `find-tag-other-frame`, the multiple-frame variant of `M-.`. See Section 22.12 [Tags], page 179.

`C-x 5 r` *filename* `RET`
: Visit file *filename* read-only, and select its buffer in another frame. This runs `find-file-read-only-other-frame`. See Section 15.2 [Visiting], page 94.

18.4 Setting Frame Parameters

This section describes commands for altering the display style and window management behavior of the selected frame.

`M-x set-foreground-color RET` *color* `RET`
: Specify color *color* for the foreground of the selected frame.

`M-x set-background-color RET` *color* `RET`
: Specify color *color* for the background of the selected frame.

`M-x set-cursor-color RET` *color* `RET`
: Specify color *color* for the cursor of the selected frame.

`M-x set-mouse-color RET` *color* `RET`
: Specify color *color* for the mouse cursor when it is over the selected frame.

`M-x set-border-color RET` *color* `RET`
: Specify color *color* for the border of the selected frame.

`M-x auto-raise-mode`
: Toggle whether or not the selected frame should auto-raise. Auto-raise means that every time you move the mouse onto the frame, it raises the frame.

`M-x auto-lower-mode`
: Toggle whether or not the selected frame should auto-lower. Auto-lower means that every time you move the mouse off of the frame, the frame moves to the bottom of the stack of X windows.

`M-x set-default-font RET` *font* `RET`
: Specify font *font* as the default for the selected frame. See Section A.6 [Font X], page 334, for ways to list the available fonts on your system.

  You can also set a frame's default font through a pop-up menu. Press `C-MOUSE-3` to activate this menu.

18.5 Scroll Bars

When using X, Emacs normally makes a *scroll bar* at the right of each Emacs window. The scroll bar runs the height of the window, and shows a moving rectangular inner box which represents the portion of the buffer currently displayed. The entire height of the scroll bar represents the entire length of the buffer.

You can use `MOUSE-2` (normally, the middle button) in the scroll bar to move or drag the inner box up and down. If you move it to the top of the scroll bar, you see the top of the buffer. If you move it to the bottom of the scroll bar, you see the bottom of the buffer.

The left and right buttons in the scroll bar scroll by controlled increments. `MOUSE-1` (normally, the left button) moves the line at the level where you click up to the top of the window. `MOUSE-3` (normally, the right button) moves the line at the top of the window down to the level where you click. By clicking repeatedly in the same place, you can scroll by the same distance over and over.

You can enable or disable Scroll Bar mode with the command `M-x scroll-bar-mode`. With no argument, it toggles the use of scroll bars. With an argument, it turns use of scroll bars on if and only if the argument is positive. This command applies to all frames, including frames yet to be created.

To enable or disable scroll bars for just the selected frame, use the `M-x toggle-scroll-bar` command.

18.6 Menu Bars

By default, each Emacs frame has a menu bar at the top which you can use to perform certain common operations. There's no need to describe them in detail here, as you can more easily see for yourself; also, we may change them and add to them in subsequent Emacs versions.

Each of the operations in the menu bar is bound to an ordinary Emacs command which you can invoke equally well with `M-x` or with its own key bindings. To see the command's name and documentation, type `C-h k` and then select the menu bar item you are interested in.

You can turn display of menu bars on or off with `M-x menu-bar-mode`. With no argument, this command toggles Menu Bar mode, a minor mode. With an argument, the command turns Menu Bar mode on if the argument is positive, off if the argument is not positive.

18.7 Using Multiple Typefaces

When using Emacs with X, you can set up multiple styles of displaying characters. The aspects of style that you can control are the type font, the foreground color, the background color, and whether to underline.

The way you control display style is by defining named *faces*. Each face can specify a type font, a foreground color, a background color, and an underline flag; but it does not have to specify all of them.

The style of display used for a given character in the text is determined by combining several faces. Which faces to use is always set up by Lisp programs, at present, by means of text properties and overlays. Any aspect of the display style that isn't specified by overlays or text properties comes from the frame itself.

To see what faces are currently defined, and what they look like, type `M-x list-faces-display`. It's possible for a given face to look different in different frames; this command shows the appearance in the frame in which you type it.

When Transient Mark mode is enabled, the text of the region is highlighted when the mark is active. This uses a face named `region`; you can control the style of highlighting by changing the style of this face with the commands below. See Section 9.1 [Setting Mark], page 51, for more information about Transient Mark mode and activation and deactivation of the mark.

One easy way to use faces is to turn on Font-Lock mode. This minor mode, which is always local to a particular buffer, arranges to choose faces according to the syntax of the text you are editing. It can recognize comments and strings in any major mode; for several major modes, it can also recognize and properly highlight various other important parts of the text. To get the full benefit of Font-Lock mode, you need to choose a default font which has bold, italic, and bold-italic variants.

18.8 Modifying Faces

Here are the commands for users to change the font of a face.

`M-x set-face-font RET` *face* `RET` *font* `RET`
: Use font *font* in face *face*.

`M-x make-face-bold RET` *face* `RET`
: Convert face *face* to use a bold version of its current font.

`M-x make-face-italic RET` *face* `RET`
: Convert face *face* to use a italic version of its current font.

`M-x make-face-bold-italic RET` *face* `RET`
: Convert face *face* to use a bold-italic version of its current font.

`M-x make-face-unbold RET` *face* `RET`
: Convert face *face* to use a non-bold version of its current font.

`M-x make-face-unitalic RET` *face* `RET`
: Convert face *face* to use a non-italic version of its current font.

Here are the commands for users to set the colors and underline flag of a face:

`M-x set-face-foreground RET` *face* `RET` *color* `RET`
: Use color *color* for the foreground of characters in face *face*.

`M-x set-face-background RET` *face* `RET` *color* `RET`
: Use color *color* for the background of characters in face *face*.

`M-x set-face-underline-p RET` *face* `RET` *flag* `RET`
: Specify whether to underline characters in face *face*.

`M-x invert-face RET` *face* `RET`
: Swap the foreground and background colors of face *face*.

18.9 Miscellaneous X Window Features

The following commands do user-level mangement of frames under a window system:

`C-z`
: To iconify the selected Emacs frame, type `C-z` (`iconify-or-deiconify-frame`). The normal meaning of `C-z`, to suspend Emacs, is not useful under a window system, so it has a different binding in that case.

  If you type this command on an Emacs icon, it deiconifies the frame.

`C-x 5 0`
: To delete the selected frame, type `C-x 5 0` (`delete-frame`). If there is more than one frame in this invocation of Emacs, the current frame is deleted.

`M-x transient-mark-mode`
: Under X Windows, when Transient Mark mode is enabled, Emacs highlights the region when the mark is active. This is the main motive for using Transient Mark mode. To toggle the state

of this mode, use the command `M-x transient-mark-mode`. See Chapter 9 [Mark], page 51.

19 Major Modes

Emacs provides many alternative *major modes*, each of which customizes Emacs for editing text of a particular sort. The major modes are mutually exclusive, and each buffer has one major mode at any time. The mode line normally shows the name of the current major mode, in parentheses (see Chapter 2 [Mode Line], page 19).

The least specialized major mode is called *Fundamental mode*. This mode has no mode-specific redefinitions or variable settings, so that each Emacs command behaves in its most general manner, and each option is in its default state. For editing any specific type of text, such as Lisp code or English text, you should switch to the appropriate major mode, such as Lisp mode or Text mode.

Selecting a major mode changes the meanings of a few keys to become more specifically adapted to the language being edited. The ones which are changed frequently are `TAB`, `DEL`, and `LFD`. In addition, the commands which handle comments use the mode to determine how comments are to be delimited. Many major modes redefine the syntactical properties of characters appearing in the buffer. See Section 31.6 [Syntax], page 311.

The major modes fall into three major groups. Lisp mode (which has several variants), C mode, Fortran mode and others are for specific programming languages. Text mode, Nroff mode, TeX mode and Outline mode are for editing English text. The remaining major modes are not intended for use on users' files; they are used in buffers created for specific purposes by Emacs, such as Dired mode for buffers made by Dired (see Chapter 28 [Dired], page 241), and Mail mode for buffers made by `C-x m` (see Chapter 26 [Sending Mail], page 221), and Shell mode for buffers used for communicating with an inferior shell process (see Section 30.3.2 [Interactive Shell], page 276).

Most programming language major modes specify that only blank lines separate paragraphs. This is so that the paragraph commands remain useful. (See Section 21.3 [Paragraphs], page 145.) They also cause Auto Fill mode to use the definition of `TAB` to indent the new lines it creates. This is because most lines in a program are usually indented. (See Chapter 20 [Indentation], page 139.)

19.1 How Major Modes are Chosen

You can select a major mode explicitly for the current buffer, but most of the time Emacs determines which mode to use based on the file name or on special text in the file.

Explicit selection of a new major mode is done with a `M-x` command. From the name of a major mode, add `-mode` to get the name of a command to select that mode. Thus, you can enter Lisp mode by executing `M-x lisp-mode`.

When you visit a file, Emacs usually chooses the right major mode based on the file's name. For example, files whose names end in '`.c`' are edited in C mode. The correspondence between file names and major mode is controlled by the variable `auto-mode-alist`. Its value is a list in which each element has the form

```
(regexp . mode-function)
```

For example, one element normally found in the list has the form `("\\.c$" . c-mode)`, and it is responsible for selecting C mode for files whose names end in '`.c`'. (Note that '`\\`' is needed in Lisp syntax to include a '`\`' in the string, which is needed to suppress the special meaning of '`.`' in regexps.) The only practical way to change this variable is with Lisp code.

You can specify which major mode should be used for editing a certain file by a special sort of text in the first nonblank line of the file. The mode name should appear in this line both preceded and followed by '`-*-`'. Other text may appear on the line as well. For example,

```
;-*-Lisp-*-
```

tells Emacs to use Lisp mode. Such an explicit specification overrides any defaulting based on the file name. Note how the semicolon is used to make Lisp treat this line as a comment.

Another format of mode specification is

```
-*-Mode: modename;-*-
```

which allows you to specify local variables as well, like this:

```
-*- mode: modename; var: value; ... -*-
```

See Section 31.2.5 [File Variables], page 297, for more information about this.

When you visit a file that does not specify a major mode to use, or when you create a new buffer with `C-x b`, the variable `default-major-mode` specifies which major mode to use. Normally its value is the symbol `fundamental-mode`, which specifies Fundamental mode. If `default-major-mode` is `nil`, the major mode is taken from the previously selected buffer.

If you change the major mode of a buffer, you can go back to the major mode Emacs would choose automatically: use the command `M-x normal-mode` to do this. This is the same function that `find-file` calls to choose the major mode. It also processes the file's local variables list if any.

20 Indentation

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

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

`LFD` Perform `RET` followed by `TAB` (`newline-and-indent`).

`M-^` Merge two lines (`delete-indentation`). This would cancel out the effect of `LFD`.

`C-M-o` Split line at point; text on the line after point becomes a new line indented to the same column that it now starts in (`split-line`).

`M-m` Move (forward or back) to the first nonblank character on the current line (`back-to-indentation`).

`C-M-\` Indent several lines to same column (`indent-region`).

`C-x TAB` Shift block of lines rigidly right or left (`indent-rigidly`).

`M-i` Indent from point to the next prespecified tab stop column (`tab-to-tab-stop`).

`M-x indent-relative`
Indent from point to under an indentation point in the previous line.

Most programming languages have some indentation convention. For Lisp code, lines are indented according to their nesting in parentheses. The same general idea is used for C code, though many details are different.

Whatever the language, to indent a line, use the `TAB` command. Each major mode defines this command to perform the sort of indentation appropriate for the particular language. In Lisp mode, `TAB` aligns the line according to its depth in parentheses. No matter where in the line you are when you type `TAB`, it aligns the line as a whole. In C mode, `TAB` implements a subtle and sophisticated indentation style that knows about many aspects of C syntax.

In Text mode, `TAB` runs the command `tab-to-tab-stop`, which indents to the next tab stop column. You can set the tab stops with `M-x edit-tab-stops`.

20.1 Indentation Commands and Techniques

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

To move over the indentation on a line, do `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 LFD`.

`C-M-o` (`split-line`) moves the text from point to the end of the line vertically down, so that the current line becomes two lines. `C-M-o` first moves point forward over any spaces and tabs. Then it inserts after point a newline and enough indentation to reach the same column point is on. Point remains before the inserted newline; in this regard, `C-M-o` resembles `C-o`.

To join two lines cleanly, use the `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 21.5.3 [Fill Prefix], page 149.

There are also commands for changing the indentation of several lines at once. `C-M-\` (`indent-region`) gives each line which begins in the region the "usual" indentation by invoking `TAB` at the beginning of the line. A numeric argument specifies the column to indent to, and each line is shifted left or right so that its first nonblank character appears in that column. `C-x TAB` (`indent-rigidly`) moves all of the lines in the region right by its argument (left, for negative arguments). The whole group of lines moves rigidly sideways, which is how the command gets its name.

`M-x indent-relative` indents at point based on the previous line (actually, the last nonempty line). It inserts whitespace at point, moving point, until it is underneath an indentation point in the previous line. An indentation point is the end of a sequence of whitespace or the end of the line. If point is farther right than any indentation point in the previous line, the whitespace before point is deleted and the first indentation point then applicable is used. If no indentation point is applicable even then, `indent-relative` runs `tab-to-tab-stop` (see next section).

`indent-relative` is the definition of `TAB` in Indented Text mode. See Chapter 21 [Text], page 143.

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. In the tab stop buffer, C-c C-c runs the function edit-tab-stops-note-changes rather than its usual definition save-buffer. edit-tab-stops records which buffer was current when you invoked it, and stores the tab stops back in that buffer; normally all buffers share the same tab stops and changing them in one buffer affects all, but if you happen to make tab-stop-list local in one buffer then edit-tab-stops in that buffer will edit the local settings.

Here is what the text representing the tab stops looks like for ordinary tab stops every eight columns.

```
        :       :       :       :       :       :
0         1         2         3         4
0123456789012345678901234567890123456789012345678
To install changes, type C-c C-c
```

The first line contains a colon at each tab stop. The remaining lines are present just to help you see where the colons are and know what to do.

Note that the tab stops that control tab-to-tab-stop have nothing to do with displaying tab characters in the buffer. See Section 12.6 [Display Vars], page 73, 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 31.2.4 [Locals], page 295.

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. For text which contains embedded commands for text formatters, Emacs has other major modes, each for a particular text formatter. Thus, for input to TEX, you would use TEX mode; for input to nroff, Nroff mode.

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`, `C-t` and `DEL`. `M-@` is related 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`', do `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 14.2 [Transpose], page 89, 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.

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 31.6 [Syntax], page 311.

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. Emacs assumes that the typist's convention is followed, and thus considers a sentence to end wherever there is a '`.`', '`?`' or '`!`' followed by the end of a line or two spaces, with any number of '`)`', '`]`', '`'`', or '`"`' characters allowed in between. A sentence also begins or ends wherever a paragraph begins or ends.

Neither `M-a` nor `M-e` moves past the newline or spaces beyond the sentence edge at which it is stopping.

Just as `C-a` and `C-e` have a kill command, `C-k`, to go with them, so `M-a` and `M-e` have a corresponding kill command `M-k` (`kill-sentence`) which kills from point to the end of the sentence. With minus one as an argument it kills back to the beginning of the sentence. Larger arguments serve as a repeat count.

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

The variable `sentence-end` controls recognition of the end of a sentence. It is a regexp that matches the last few characters of a sentence, together with the whitespace following the sentence. Its normal value is

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

This example is explained in the section on regexps. See Section 13.5 [Regexps], page 79.

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 part of any paragraph. Also, an indented line starts a new paragraph.

In major modes for programs (as opposed to Text mode), paragraphs begin and end only at blank lines. This makes the paragraph commands continue to be useful even though there are no paragraphs per se.

When there is a fill prefix, then paragraphs are delimited by all lines which don't start with the fill prefix. See Section 21.5 [Filling], page 147.

When you wish to operate on a paragraph, you can use the command `M-h` (`mark-paragraph`) to set the region around it. This command puts point at the beginning and mark at the end of the paragraph point was in. If point is between paragraphs (in a run of blank lines, or at a boundary), the paragraph following point is surrounded by point and mark. If there are blank lines preceding the first line of the paragraph, one of these blank lines is included in the region. Thus, for example, `M-h C-w` kills the paragraph around or after point.

The precise definition of a paragraph boundary is controlled by the variables `paragraph-separate` and `paragraph-start`. The value of `paragraph-start` is a regexp that should match any line that either starts or separates paragraphs. The value of `paragraph-separate` is another regexp that should match only lines that separate paragraphs without being part of any paragraph. Lines that start a new paragraph and are contained in it must match only `paragraph-start`, not `paragraph-separate`. For example, normally `paragraph-start` is `"^[ \t\n\f]"` and `paragraph-separate` is `"^[ \t\f]*$"`.

Normally it is desirable for page boundaries to separate paragraphs. The default values of these variables recognize the usual separator for pages.

21.4 Pages

Files are often thought of as divided into *pages* by the *formfeed* character (ASCII control-L, octal code 014). For example, if a file is printed on a line printer, each page of the file, in this sense, will start on a new page of paper. Emacs treats a page-separator character just like any other character. You can insert it with `C-q C-l`, or 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). This command can be followed by `C-w` to kill a page which is to be moved elsewhere. If it is inserted after a page delimiter, at a place where `C-x ]` or `C-x [` would take you, then the page will be properly delimited before and after once again.

A numeric argument to `C-x C-p` is used to specify which page to go to, relative to the current one. Zero means the current page. One means the next page, and −1 means the previous one.

The `C-x l` command (`count-lines-page`) is good for deciding where to break a page in two. It prints in the echo area the total number of lines in the current page, and then divides it up into those preceding the current line and those following, as in

```
Page has 96 (72+25) lines
```

Notice that the sum is off by one; this is correct if point is not at the beginning of a line.

The variable `page-delimiter` 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

With Auto Fill mode, text can be *filled* (broken up into lines that fit in a specified width) as you insert it. If you alter existing text it may no longer be properly filled; then you can use the explicit fill commands to fill the paragraph again.

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, turned on or off for each buffer individually. See Section 31.1 [Minor Modes], page 291.

In Auto Fill mode, lines are broken automatically at spaces when they get longer than the desired width. Line breaking and rearrangement takes place only when you type SPC or RET. If you wish to insert a space or newline without permitting line-breaking, type C-q SPC or C-q LFD (recall that a newline is really a linefeed). Also, C-o inserts a newline without line breaking.

Auto Fill mode works well with Lisp mode, because when it makes a new line in Lisp mode it indents that line with TAB. If a line ending in a comment gets too long, the text of the comment is split into two comment lines. Optionally new comment delimiters are inserted at the end of the first line and the beginning of the second so that each line is a separate comment; the variable comment-multi-line controls the choice (see Section 22.7.2 [Comments], page 175).

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 31.7 [Init File], page 311.

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.

The command `M-s` (`center-line`) centers the current line within the current fill column. With an argument, it centers several lines individually and moves past them.

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 21.3 [Paragraphs], page 145). 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`.)

When `adaptive-fill-mode` is non-`nil` (which is normally the case), if you use `fill-region-as-paragraph` on an indented paragraph and you don't have a fill prefix, it uses the indentation of the second line of the paragraph as the fill prefix. The effect of adaptive filling is not noticeable in Text mode, because an indented line counts as a paragraph starter and thus each line of an indented paragraph is considered a paragraph of its own. But you do notice the effect in Indented Text mode and some other major modes.

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 31.2.4 [Locals], page 295.

The easiest way to set `fill-column` is to use the command `C-x f` (`set-fill-column`). With no argument, it sets `fill-column` to the current horizontal position of point. With a numeric argument, it uses that as the new fill column.

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), use the *fill prefix* feature. The fill prefix is a string which Emacs expects every line to start with, and which is not included in filling.

`C-x .`
: Set the fill prefix (`set-fill-prefix`).

`M-q`
: Fill a paragraph using current fill prefix (`fill-paragraph`).

`M-x fill-individual-paragraphs`
: Fill the region, considering each change of indentation as starting a new paragraph.

`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. The fill prefix is also inserted on new lines made automatically by Auto Fill mode. Lines that do not start with the fill prefix are considered to start paragraphs, both in `M-q` and the paragraph commands; this is just right if you are using paragraphs with hanging indentation (every line indented except the first one). Lines which are blank or indented once the prefix is removed also separate or start paragraphs; this is what you want if you are writing multi-paragraph comments with a comment delimiter on each line.

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.
```

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 4.6 [Blank Lines], page 29). Conversely, the command `M-^` deletes the prefix (if it occurs) after the newline that it deletes (see Chapter 20 [Indentation], page 139).

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.

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 31.2.4 [Locals], page 295.

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 31.4.9 [Disabling], page 310.

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. Invoke `M-x text-mode` to enter Text mode. In Text mode, `TAB` runs the function `tab-to-tab-stop`, which allows you to use arbitrary tab stops set with `M-x edit-tab-stops` (see Section 20.2 [Tab Stops], page 141). Features concerned with comments in programs are turned off except when explicitly invoked. The syntax table is changed so that periods are not considered part of a word, while apostrophes, backspaces and underlines are.

A similar variant mode is Indented Text mode, intended for editing text in which most lines are indented. This mode defines `TAB` to run `indent-relative` (see Chapter 20 [Indentation], page 139), and makes Auto Fill indent the lines it creates. The result is that normally a line made by Auto Filling, or by `LFD`, is indented just like the previous line. In Indented Text mode, only blank lines separate paragraphs—indented lines continue the current paragraph. Use `M-x indented-text-mode` to select 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 14.4 [Spelling], page 90.

Entering Text mode or Indented 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 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 31.2.3 [Hooks], page 294.

21.8 Outline Mode

Outline mode is a major mode much like Text mode but intended for editing outlines. It allows you to make parts of the text temporarily invisible so that you can see just the overall structure of the outline. Type `M-x outline-mode` to switch to Outline mode as the major mode of the current buffer. Type `M-x outline-minor-mode` to enable Outline mode as a minor mode in the current buffer. When Outline minor mode is enabled, the `C-c` commands of Outline mode replace those of the major mode.

When a line is invisible in outline mode, it does not appear on the screen. The screen appears exactly as if the invisible line were deleted, except that an ellipsis (three periods in a row) appears at the end of the previous visible line (only one ellipsis no matter how many invisible lines follow).

All editing commands treat the text of the invisible line as part of the previous visible line. For example, `C-n` moves onto the next visible line.

Killing an entire visible line, including its terminating newline, really kills all the following invisible lines along with it; yanking it all back yanks the invisible lines and they remain invisible.

Entering Outline mode runs the hook `text-mode-hook` followed by the hook `outline-mode-hook` (see Section 31.2.3 [Hooks], page 294).

21.8.1 Format of Outlines

Outline mode assumes that the lines in the buffer are of two types: *heading lines* and *body lines*. A heading line represents a topic in the outline. Heading lines start with one or more stars; the number of stars determines the depth of the heading in the outline structure. Thus, a heading line with one star is a major topic; all the heading lines with two stars between it and the next one-star heading are its subtopics; and so on. Any line that is not a heading line is a body line. Body lines belong with the preceding heading line. Here is an example:

```
* Food

This is the body,
which says something about the topic of food.

** Delicious Food

This is the body of the second-level header.

** Distasteful Food

This could have
a body too, with
several lines.

*** Dormitory Food

* Shelter

A second first-level topic with its header line.
```

A heading line together with all following body lines is called collectively an *entry*. A heading line together with all following deeper heading lines and their body lines is called a *subtree*.

You can customize the criterion for distinguishing heading lines by setting the variable `outline-regexp`. Any line whose beginning has a match for this regexp is considered a heading line. Matches that start within a line (not at the beginning) do not count. The length of the matching text determines the level of the heading; longer matches make a more deeply nested level. Thus, for example, if a text formatter has commands '`@chapter`', '`@section`' and '`@subsection`' to divide the document into chapters and sections, you could make those lines count as heading lines by setting `outline-regexp` to '`"@chap\\|@\\(sub\\)*section"`'. Note the trick: the two words '`chapter`' and '`section`' are equally long, but by defining the regexp to match only '`chap`' we ensure that the length of the text matched on a chapter heading is shorter, so that Outline mode will know that sections are contained in chapters. This works as long as no other command starts with '`@chap`'.

Outline mode makes a line invisible by changing the newline before it into an ASCII control-M (code 015). Most editing commands that work on lines treat an invisible line as part of the previous line because, strictly speaking, it *is* part of that line, since there is no longer a newline in between. When you save the file in Outline mode, control-M characters are saved as newlines, so the invisible lines become ordinary lines in the file. But saving does not change the visibility status of a line inside Emacs.

21.8.2 Outline Motion Commands

There are some special motion commands in Outline mode that move backward and forward to heading lines.

`C-c C-n`
: Move point to the next visible heading line (`outline-next-visible-heading`).

`C-c C-p`
: Move point to the previous visible heading line (`outline-previous-visible-heading`).

`C-c C-f`
: Move point to the next visible heading line at the same level as the one point is on (`outline-forward-same-level`).

`C-c C-b`
: Move point to the previous visible heading line at the same level (`outline-backward-same-level`).

`C-c C-u`
: Move point up to a lower-level (more inclusive) visible heading line (`outline-up-heading`).

`C-c C-n` (`next-visible-heading`) moves down to the next heading line. `C-c C-p` (`previous-visible-heading`) moves similarly backward. Both accept numeric arguments as repeat counts. The names emphasize that invisible headings are skipped, but this is not really a special feature. All editing commands that look for lines ignore the invisible lines automatically.

More powerful motion commands understand the level structure of headings. `C-c C-f` (`outline-forward-same-level`) and `C-c C-b` (`outline-backward-same-level`) move from one heading line to another visible heading at the same depth in the outline. `C-c C-u` (`outline-up-heading`) moves backward to another heading that is less deeply nested.

21.8.3 Outline Visibility Commands

The other special commands of outline mode are used to make lines visible or invisible. Their names all start with `hide` or `show`. Most of them fall into pairs of opposites. They are not undoable; instead, you can undo right past them. Making lines visible or invisible is simply not recorded by the undo mechanism.

`M-x hide-body`
: Make all body lines in the buffer invisible.

`M-x show-all`
: Make all lines in the buffer visible.

`C-c C-h`
: Make everything under this heading invisible, not including this heading itself (`hide-subtree`).

`C-c C-s`
: Make everything under this heading visible, including body, subheadings, and their bodies (`show-subtree`).

`M-x hide-leaves`
: Make the body of this heading line, and of all its subheadings, invisible.

`M-x show-branches`
: Make all subheadings of this heading line, at all levels, visible.

`C-c C-i`
: Make immediate subheadings (one level down) of this heading line visible (`show-children`).

`M-x hide-entry`
: Make this heading line's body invisible.

`M-x show-entry`
: Make this heading line's body visible.

Two commands that are exact opposites are `M-x hide-entry` and `M-x show-entry`. They are used with point on a heading line, and apply only to the body lines of that heading. The subtopics and their bodies are not affected.

Two more powerful opposites are `C-c C-h` (`hide-subtree`) and `C-c C-s` (`show-subtree`). Both expect to be used when point is on a heading line, and both apply to all the lines of that heading's *subtree*: its body, all its subheadings, both direct and indirect, and all of their bodies. In other words, the subtree contains everything following this heading line, up to and not including the next heading of the same or higher rank.

Intermediate between a visible subtree and an invisible one is having all the subheadings visible but none of the body. There are two commands for doing this, depending on whether you want to hide the bodies or make the subheadings visible. They are `M-x hide-leaves` and `M-x show-branches`.

A little weaker than `show-branches` is `C-c C-i` (`show-children`). It makes just the direct subheadings visible—those one level down. Deeper subheadings remain invisible, if they were invisible.

Two commands have a blanket effect on the whole file. `M-x hide-body` makes all body lines invisible, so that you see just the outline structure. `M-x show-all` makes all lines visible. These commands can be thought of as a pair of opposites even though `M-x show-all` applies to more than just body lines.

You can turn off the use of ellipses at the ends of visible lines by setting `selective-display-ellipses` to `nil`. Then there is no visible indication of the presence of invisible lines.

21.9 TeX Mode

TeX is a powerful text formatter written by Donald Knuth; it is also free, like GNU Emacs. LaTeX is a simplified input format for TeX, implemented by TeX macros; it comes with TeX. SliTeX is a special form of LaTeX.

Emacs has a special TeX mode for editing TeX input files. It provides facilities for checking the balance of delimiters and for invoking TeX on all or part of the file.

TeX mode has three variants, Plain TeX mode, LaTeX mode, and SliTeX mode (these three distinct major modes differ only slightly). They are designed for editing the three different formats. The command `M-x tex-mode` looks at the contents of the buffer to determine whether the contents appear to be either LaTeX input or SliTeX input; it then selects the appropriate mode. If it can't tell which is right (e.g., the buffer is empty), the variable `tex-default-mode` controls which mode is used.

When `M-x tex-mode` does not guess right, you can use the commands `M-x plain-tex-mode`, `M-x latex-mode`, and `M-x slitex-mode` to select explicitly the particular variants of TeX mode.

21.9.1 TeX Editing Commands

Here are the special commands provided in TeX mode for editing the text of the file.

`"`
: Insert, according to context, either ‘``’ or ‘"’ or ‘''’ (`tex-insert-quote`).

`LFD`
: Insert a paragraph break (two newlines) and check the previous paragraph for unbalanced braces or dollar signs (`tex-terminate-paragraph`).

`M-x validate-tex-region`
: Check each paragraph in the region for unbalanced braces or dollar signs.

`C-c {`
: Insert '`{}`' and position point between them (`tex-insert-braces`).

`C-c }`
: Move forward past the next unmatched close brace (`up-list`).

In TeX, the character '`"`' is not normally used; we use '` `` `' to start a quotation and '`''`' to end one. To make editing easier under this formatting convention, TeX mode overrides the normal meaning of the key `"` with a command that inserts a pair of single-quotes or backquotes (`tex-insert-quote`). To be precise, this command inserts '` `` `' after whitespace or an open brace, '`"`' after a backslash, and '`''`' after any other character.

If you need the character '`"`' itself in unusual contexts, use `C-q` to insert it. Also, `"` with a numeric argument always inserts that number of '`"`' characters.

In TeX mode, '`$`' has a special syntax code which attempts to understand the way TeX math mode delimiters match. When you insert a '`$`' that is meant to exit math mode, the position of the matching '`$`' that entered math mode is displayed for a second. This is the same feature that displays the open brace that matches a close brace that is inserted. However, there is no way to tell whether a '`$`' enters math mode or leaves it; so when you insert a '`$`' that enters math mode, the previous '`$`' position is shown as if it were a match, even though they are actually unrelated.

TeX uses braces as delimiters that must match. Some users prefer to keep braces balanced at all times, rather than inserting them singly. Use `C-c {` (`tex-insert-braces`) to insert a pair of braces. It leaves point between the two braces so you can insert the text that belongs inside. Afterward, use the command `C-c }` (`up-list`) to move forward past the close brace.

There are two commands for checking the matching of braces. `LFD` (`tex-terminate-paragraph`) checks the paragraph before point, and inserts two newlines to start a new paragraph. It prints a message in the echo area if any mismatch is found. `M-x validate-tex-region` checks a region, paragraph by paragraph. When it finds a paragraph that contains a mismatch, it displays point at the beginning of the paragraph for a few seconds and pushes a mark at that spot. Scanning continues until the whole buffer has been checked or until you type another key. The positions of the last several paragraphs with mismatches can be found in the mark ring (see Chapter 10 [Mark Ring], page 57).

Note that Emacs commands count square brackets and parentheses in TeX mode, not just braces. This is not strictly correct for the purpose of checking TeX syntax. However, parentheses and square brackets are likely to be used in text as matching delimiters and it is useful for the various motion commands and automatic match display to work with them.

21.9.2 LaTeX Editing Commands

LaTeX mode provides a few extra features not applicable to plain TeX.

`C-c C-o` Insert '`\begin`' and '`\end`' for LaTeX block and position point on a line between them. (`tex-latex-block`).

`C-c C-e` Close the last unended block for LaTeX (`tex-close-latex-block`).

In LaTeX input, '`\begin`' and '`\end`' commands are used to group blocks of text. To insert a '`\begin`' and a matching '`\end`' (on a new line following the '`\begin`'), use `C-c C-o` (`tex-latex-block`). A blank line is inserted between the two, and point is left there.

Emacs knows all of the standard LaTeX block names and will permissively complete a partially entered block name (see Section 6.3 [Completion], page 37). You can add your own list of block names to those known by Emacs with the variable `latex-block-names`. For example, to add '`theorem`', '`corollary`', and '`proof`', include the line

```
(setq latex-block-names '("theorem" "corollary" "proof"))
```

to your '`.emacs`' file.

In LaTeX input, '`\begin`' and '`\end`' commands must balance. You can use `C-c C-e` (`tex-close-latex-block`) to insert automatically a matching '`\end`' to match the last unmatched '`\begin`'. The '`\end`' will be indented to match the corresponding '`\begin`'. The '`\end`' will be followed by a newline if point is at the beginning of a line.

21.9.3 TeX Printing Commands

You can invoke TeX as an inferior of Emacs on either the entire contents of the buffer or just a region at a time. Running TeX in this way on just one chapter is a good way to see what your changes look like without taking the time to format the entire file.

`C-c C-r` Invoke TeX on the current region, together with the buffer's header (`tex-region`).

`C-c C-b` Invoke TeX on the entire current buffer (`tex-buffer`).

`C-c TAB` Invoke BibTeX on the current file (`tex-bibtex-file`).

`C-c C-f` Invoke TeX on the current file (`tex-file`).

`C-c C-l` Recenter the window showing output from the inferior TeX so that the last line can be seen (`tex-recenter-output-buffer`).

`C-c C-k` Kill the TeX subprocess (`tex-kill-job`).

`C-c C-p` Print the output from the last `C-c C-r`, `C-c C-b`, or `C-c C-f` command (`tex-print`).

`C-c C-v` Preview the output from the last `C-c C-r`, `C-c C-b`, or `C-c C-f` command (`tex-view`).

`C-c C-q` Show the printer queue (`tex-show-print-queue`).

You can pass the current buffer through an inferior TEX by means of `C-c C-b` (`tex-buffer`). The formatted output appears in a temporary; to print it, type `C-c C-p` (`tex-print`). Afterward use `C-c C-q` (`tex-show-print-queue`) to view the progress of your output towards being printed. If your terminal has the ability to display TEX output files, you can preview the output on the terminal with `C-c C-v` (`tex-view`).

You can specify the directory to use for running TEX by setting the variable `tex-directory`. `"."` is the default value. If your environment variable `TEXINPUTS` contains relative directory names, or if your files contains '`\input`' commands with relative file names, then `tex-directory` *must* be `"."` or you will get the wrong results. Otherwise, it is safe to specify some other directory, such as '`/tmp`'.

If you want to specify which shell commands are used in the inferior TEX, you can do so by setting the values of the variables `tex-run-command`, `latex-run-command`, `slitex-run-command`, `tex-dvi-print-command`, `tex-dvi-view-command`, and `tex-show-queue-command`. You *must* set the value of `tex-dvi-view-command` for your particular terminal; this variable has no default value. The other variables have default values that may (or may not) be appropriate for your system.

Normally, the file name given to these commands comes at the end of the command string; for example, '`latex` *filename*'. In some cases, however, the file name needs to be embedded in the command; an example is when you need to provide the file name as an argument to one command whose output is piped to another. You can specify where to put the file name with '`*`' in the command string. For example,

```
(setq tex-dvi-print-command "dvips -f * | lpr")
```

The terminal output from TEX, including any error messages, appears in a buffer called '`*tex-shell*`'. If TEX gets an error, you can switch to this buffer and feed it input (this works as in Shell mode; see Section 30.3.2 [Interactive Shell], page 276). Without switching to this buffer you can scroll it so that its last line is visible by typing `C-c C-l`.

Type `C-c C-k` (`tex-kill-job`) to kill the TEX process if you see that its output is no longer useful. Using `C-c C-b` or `C-c C-r` also kills any TEX process still running.

You can also pass an arbitrary region through an inferior TEX by typing `C-c C-r` (`tex-region`). This is tricky, however, because most files of

TeX input contain commands at the beginning to set parameters and define macros, without which no later part of the file will format correctly. To solve this problem, `C-c C-r` allows you to designate a part of the file as containing essential commands; it is included before the specified region as part of the input to TeX. The designated part of the file is called the *header*.

To indicate the bounds of the header in Plain TeX mode, you insert two special strings in the file. Insert '`%**start of header`' before the header, and '`%**end of header`' after it. Each string must appear entirely on one line, but there may be other text on the line before or after. The lines containing the two strings are included in the header. If '`%**start of header`' does not appear within the first 100 lines of the buffer, `C-c C-r` assumes that there is no header.

In LaTeX mode, the header begins with '`\documentstyle`' and ends with '`\begin{document}`'. These are commands that LaTeX requires you to use in any case, so nothing special needs to be done to identify the header.

The commands (`tex-buffer`) and (`tex-region`) do all of their work in a temporary directory, and do not have available any of the auxiliary files needed by TeX for cross-references; these commands are generally not suitable for running the final copy in which all of the cross-references need to be correct. When you want the auxiliary files, use `C-c C-f` (`tex-file`) which runs TeX on the current buffer's file, in that file's directory. Before TeX runs, you will be asked about saving any modified buffers. Generally, you need to use (`tex-file`) twice to get cross-references correct.

For LaTeX files, you can use BibTeX to process the auxiliary file for the current buffer's file. BibTeX looks up bibliographic citations in a data base and prepares the cited references for the bibliography section. The command `C-c TAB` (`tex-bibtex-file`) runs the shell command (`tex-bibtex-command`) to produce a '`.bbl`' file for the current buffer's file. Generally, you need to do `C-c C-f` (`tex-file`) once to generate the '`.aux`' file, then do `C-c TAB` (`tex-bibtex-file`), and then repeat `C-c C-f` (`tex-file`) twice more to get the cross-references correct.

Entering any kind of TeX mode runs the hooks `text-mode-hook` and `tex-mode-hook`. Then it runs either `plain-tex-mode-hook` or `latex-mode-hook`, whichever is appropriate. For SliTeX files, it calls `slitex-mode-hook`. Starting the TeX shell runs the hook `tex-shell-hook`. See Section 31.2.3 [Hooks], page 294.

21.9.4 Unix TeX Distribution

TeX for Unix systems can be obtained from the University of Washington for a distribution fee.

To order a full distribution, send $200.00 for a 1/2-inch 9-track 1600 bpi (tar or cpio) tape reel, or $210.00 for a 1/4-inch 4-track QIC-24 (tar or cpio) cartridge, payable to the University of Washington to:

Northwest Computing Support Center
DR-10, Thomson Hall 35
University of Washington
Seattle, Washington 98195

Purchase orders are acceptable, but there is an extra charge of $10.00, to pay for processing charges.

For overseas orders please add $20.00 to the base cost for shipment via air parcel post, or $30.00 for shipment via courier.

The normal distribution is a tar tape, blocked 20, 1600 bpi, on an industry standard 2400 foot half-inch reel. The physical format for the 1/4 inch streamer cartridges uses QIC-11, 8000 bpi, 4-track serpentine recording for the SUN. Also, System V tapes can be written in cpio format, blocked 5120 bytes, ASCII headers.

21.10 Nroff Mode

Nroff mode is a mode like Text mode but modified to handle nroff commands present in the text. Invoke `M-x nroff-mode` to enter this mode. It differs from Text mode in only a few ways. All nroff command lines are considered paragraph separators, so that filling will never garble the nroff commands. Pages are separated by '`.bp`' commands. Comments start with backslash-doublequote. Also, three special commands are provided that are not in Text mode:

`M-n`
: Move to the beginning of the next line that isn't an nroff command (`forward-text-line`). An argument is a repeat count.

`M-p`
: Like `M-n` but move up (`backward-text-line`).

`M-?`
: Prints in the echo area the number of text lines (lines that are not nroff commands) in the region (`count-text-lines`).

The other feature of Nroff mode is that you can turn on Electric Nroff mode. This is a minor mode that you can turn on or off with `M-x electric-nroff-mode` (see Section 31.1 [Minor Modes], page 291). When the mode is on, each time you use `RET` to end a line that contains an nroff command that opens a kind of grouping, the matching nroff command to close that grouping is automatically inserted on the following line. For example, if you are at the beginning of a line and type `. ( b RET`, this inserts the matching command '`.)b`' on a new line following point.

Entering Nroff mode runs the hook `text-mode-hook`, followed by the hook `nroff-mode-hook` (see Section 31.2.3 [Hooks], page 294).

22 Editing Programs

Emacs has many commands designed to understand the syntax of programming languages such as Lisp and C. These commands can

- Move over or kill balanced expressions or *sexps* (see Section 22.2 [Lists], page 164).
- Move over or mark top-level expressions—*defuns*, in Lisp; functions, in C (see Section 22.4 [Defuns], page 166).
- Show how parentheses balance (see Section 22.6 [Matching], page 173).
- Insert, kill or align comments (see Section 22.7.2 [Comments], page 175).
- Follow the usual indentation conventions of the language (see Section 22.5 [Program Indent], page 167).

The commands for words, sentences and paragraphs are very useful in editing code even though their canonical application is for editing human language text. Most symbols contain words (see Section 21.1 [Words], page 143); sentences can be found in strings and comments (see Section 21.2 [Sentences], page 144). Paragraphs per se don't exist in code, but the paragraph commands are useful anyway, because programming language major modes define paragraphs to begin and end at blank lines (see Section 21.3 [Paragraphs], page 145). Judicious use of blank lines to make the program clearer will also provide interesting chunks of text for the paragraph commands to work on.

The selective display feature is useful for looking at the overall structure of a function (see Section 12.3 [Selective Display], page 71). This feature causes only the lines that are indented less than a specified amount to appear on the screen.

22.1 Major Modes for Programming Languages

Emacs also has major modes for the programming languages Lisp, Scheme (a variant of Lisp), Awk, C, C++, Perl, Icon, Fortran and Muddle (another variant of Lisp). There is also a major mode for makefiles, called Makefile mode.

Ideally, a major mode should be implemented for each programming language that you might want to edit with Emacs; but often the mode for one language can serve for other syntactically similar languages. The language modes that exist are those that someone decided to take the trouble to write.

There are several forms of Lisp mode, which differ in the way they interface to Lisp execution. See Section 23.3 [Executing Lisp], page 204.

Each of the programming language modes defines the `TAB` key to run an indentation function that knows the indentation conventions of that language and updates the current line's indentation accordingly. For example, in C mode `TAB` is bound to `c-indent-line`. `LFD` is normally defined to do `RET` followed by `TAB`; thus, it too indents in a mode-specific fashion.

In most programming languages, indentation is likely to vary from line to line. So the major modes for those languages rebind `DEL` to treat a tab as if it were the equivalent number of spaces (using the command `backward-delete-char-untabify`). This makes it possible to rub out indentation one column at a time without worrying whether it is made up of spaces or tabs. Use `C-b C-d` to delete a tab character before point, in these modes.

Programming language modes define paragraphs to be separated only by blank lines, so that the paragraph commands remain useful. Auto Fill mode, if enabled in a programming language major mode, indents the new lines which it creates.

Turning on a major mode runs a normal hook called the *mode hook*, which is the value of a Lisp variable. For example, turning on C mode runs the hook `c-mode-hook`. Mode hook variables for other programming language modes include `lisp-mode-hook`, `emacs-lisp-mode-hook`, `lisp-interaction-mode-hook`, `scheme-mode-hook` and `muddle-mode-hook`. See Section 31.2.3 [Hooks], page 294.

22.2 Lists and Sexps

By convention, Emacs keys for dealing with balanced expressions are usually Control-Meta characters. They tend to be analogous in function to their Control and Meta equivalents. These commands are usually thought of as pertaining to expressions in programming languages, but can be useful with any language in which some sort of parentheses exist (including human languages).

These commands fall into two classes. Some deal only with *lists* (parenthetical groupings). They see nothing except parentheses, brackets, braces (whichever ones must balance in the language you are working with), and escape characters that might be used to quote those.

The other commands deal with expressions or *sexps*. The word 'sexp' is derived from *s-expression*, the ancient term for an expression in Lisp. But in Emacs, the notion of 'sexp' is not limited to Lisp. It refers to an expression in whatever language your program is written in. Each programming language has its own major mode, which customizes the syntax tables so that expressions in that language count as sexps.

Sexps typically include symbols, numbers, and string constants, as well as anything contained in parentheses, brackets or braces.

In languages that use prefix and infix operators, such as C, it is not possible for all expressions to be sexps. For example, C mode does not recognize '`foo + bar`' as a sexp, even though it *is* a C expression; it recognizes '`foo`' as one sexp and '`bar`' as another, with the '`+`' as punctuation between them. This is a fundamental ambiguity: both '`foo + bar`' and '`foo`' are legitimate choices for the sexp to move over if point is at the '`f`'. Note that '`(foo + bar)`' is a single sexp in C mode.

Some languages have obscure forms of syntax for expressions that nobody has bothered to make Emacs understand properly.

22.3 List And Sexp Commands

`C-M-f` Move forward over a sexp (`forward-sexp`).

`C-M-b` Move backward over a sexp (`backward-sexp`).

`C-M-k` Kill sexp forward (`kill-sexp`).

`C-M-u` Move up and backward in list structure (`backward-up-list`).

`C-M-d` Move down and forward in list structure (`down-list`).

`C-M-n` Move forward over a list (`forward-list`).

`C-M-p` Move backward over a list (`backward-list`).

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

`C-M-@` Put mark after following expression (`mark-sexp`).

To move forward over a sexp, use `C-M-f` (`forward-sexp`). If the first significant character after point is an opening delimiter ('`(`' in Lisp; '`(`', '`[`' or '`{`' in C), `C-M-f` moves past the matching closing delimiter. If the character begins a symbol, string, or number, `C-M-f` moves over that.

The command `C-M-b` (`backward-sexp`) moves backward over a sexp. The detailed rules are like those above for `C-M-f`, but with directions reversed. If there are any prefix characters (single-quote, backquote and comma, in Lisp) preceding the sexp, `C-M-b` moves back over them as well. The sexp commands move across comments as if they were whitespace in most modes.

`C-M-f` or `C-M-b` with an argument repeats that operation the specified number of times; with a negative argument, it moves in the opposite direction.

Killing a sexp at a time can be done with `C-M-k` (`kill-sexp`). `C-M-k` kills the characters that `C-M-f` would move over.

The *list commands* move over lists like the sexp commands but skip blithely over any number of other kinds of sexps (symbols, strings, etc). They are `C-M-n` (`forward-list`) and `C-M-p` (`backward-list`). The main reason they are useful is that they usually ignore comments (since the comments usually do not contain any lists).

`C-M-n` and `C-M-p` stay at the same level in parentheses, when that's possible. To move *up* one (or *n*) levels, use `C-M-u` (`backward-up-list`). `C-M-u` moves backward up past one unmatched opening delimiter. A positive argument serves as a repeat count; a negative argument reverses direction of motion and also requests repetition, so it moves forward and up one or more levels.

To move *down* in list structure, use `C-M-d` (`down-list`). In Lisp mode, where '`(`' is the only opening delimiter, this is nearly the same as searching for a '`(`'. An argument specifies the number of levels of parentheses to go down.

A somewhat random-sounding command which is nevertheless handy is `C-M-t` (`transpose-sexps`), which drags the previous sexp across the next one. An argument serves as a repeat count, and a negative argument drags backwards (thus canceling out the effect of `C-M-t` with a positive argument). An argument of zero, rather than doing nothing, transposes the sexps ending after point and the mark.

To make the region be the next sexp in the buffer, use `C-M-@` (`mark-sexp`) which sets mark at the same place that `C-M-f` would move to. `C-M-@` takes arguments like `C-M-f`. In particular, a negative argument is useful for putting the mark at the beginning of the previous sexp.

The list and sexp commands' understanding of syntax is completely controlled by the syntax table. Any character can, for example, be declared to be an opening delimiter and act like an open parenthesis. See Section 31.6 [Syntax], page 311.

22.4 Defuns

In Emacs, a parenthetical grouping at the top level in the buffer is called a *defun*. The name derives from the fact that most top-level lists in a Lisp file are instances of the special form `defun`, but any top-level parenthetical grouping counts as a defun in Emacs parlance regardless of what its contents are, and regardless of the programming language in use. For example, in C, the body of a function definition is a defun.

`C-M-a` Move to beginning of current or preceding defun (`beginning-of-defun`).

`C-M-e` Move to end of current or following defun (`end-of-defun`).

`C-M-h` Put region around whole current or following defun (`mark-defun`).

The commands to move to the beginning and end of the current defun are `C-M-a` (`beginning-of-defun`) and `C-M-e` (`end-of-defun`).

If you wish to operate on the current defun, use `C-M-h` (`mark-defun`) which puts point at the beginning and mark at the end of the current or next defun. For example, this is the easiest way to get ready to move the defun to a different place in the text. In C mode, `C-M-h` runs the function `mark-c-function`, which is almost the same as `mark-defun`; the difference is that it backs up over the argument declarations, function name and returned data type so that the entire C function is inside the region. See Section 9.4 [Marking Objects], page 53.

Emacs assumes that any open-parenthesis found in the leftmost column is the start of a defun. Therefore, **never put an open-parenthesis at the left margin in a Lisp file unless it is the start of a top level list. Never put an open-brace or other opening delimiter at the beginning of a line of C code unless it starts the body of a function.** The most likely problem case is when you want an opening delimiter at the start of a line inside a string. To avoid trouble, put an escape character ('`\`', in C and Emacs Lisp, '`/`' in some other Lisp dialects) before the opening delimiter. It will not affect the contents of the string.

In the remotest past, the original Emacs found defuns by moving upward a level of parentheses until there were no more levels to go up. This always required scanning all the way back to the beginning of the buffer, even for a small function. To speed up the operation, Emacs was changed to assume that any '`(`' (or other character assigned the syntactic class of opening-delimiter) at the left margin is the start of a defun. This heuristic is nearly always right and avoids the costly scan; however, it mandates the convention described above.

22.5 Indentation for Programs

The best way to keep a program properly indented is to use Emacs to re-indent it as you change it. Emacs has commands to indent properly either a single line, a specified number of lines, or all of the lines inside a single parenthetical grouping.

22.5.1 Basic Program Indentation Commands

`TAB` Adjust indentation of current line.

`LFD` Equivalent to `RET` followed by `TAB` (`newline-and-indent`).

The basic indentation command is `TAB`, which gives the current line the correct indentation as determined from the previous lines. The function that `TAB` runs depends on the major mode; it is `lisp-indent-line` in Lisp mode, `c-indent-line` in C mode, etc. These functions understand different syntaxes for different languages, but they all do about the same thing. `TAB` in any programming language major mode inserts or deletes whitespace at the beginning of the current line, independent of where point is in the line. If point is inside the whitespace at the beginning of the line, `TAB` leaves it at the end of that whitespace; otherwise, `TAB` leaves point fixed with respect to the characters around it.

Use `C-q TAB` to insert a tab at point.

When entering a large amount of new code, use `LFD` (`newline-and-indent`), which is equivalent to a `RET` followed by a `TAB`. `LFD` creates a blank line, and then gives it the appropriate indentation.

`TAB` indents the second and following lines of the body of a parenthetical grouping each under the preceding one; therefore, if you alter one line's indentation to be nonstandard, the lines below will tend to follow it. This is the right behavior in cases where the standard result of `TAB` is unaesthetic.

Remember that an open-parenthesis, open-brace or other opening delimiter at the left margin is assumed by Emacs (including the indentation routines) to be the start of a function. Therefore, you must never have an opening delimiter in column zero that is not the beginning of a function, not even inside a string. This restriction is vital for making the indentation commands fast; you must simply accept it. See Section 22.4 [Defuns], page 166, for more information on this.

22.5.2 Indenting Several Lines

When you wish to re-indent several lines of code which have been altered or moved to a different level in the list structure, you have several commands available.

`C-M-q`
: Re-indent all the lines within one list (`indent-sexp`).

`C-u TAB`
: Shift an entire list rigidly sideways so that its first line is properly indented.

`C-M-\`
: Re-indent all lines in the region (`indent-region`).

You can re-indent the contents of a single list by positioning point before the beginning of it and typing `C-M-q` (`indent-sexp` in Lisp mode, `indent-c-exp` in C mode; also bound to other suitable commands in other modes). The indentation of the line the sexp starts on is not changed; therefore, only the relative indentation within the list, and not its position, is changed. To correct the position as well, type a `TAB` before the `C-M-q`.

If the relative indentation within a list is correct but the indentation of its beginning is not, go to the line the list begins on and type `C-u TAB`. When `TAB` is given a numeric argument, it moves all the lines in the grouping starting on the current line sideways the same amount that the current line moves. It is clever, though, and does not move lines that start inside strings, or C preprocessor lines when in C mode.

Another way to specify the range to be re-indented is with point and mark. The command `C-M-\` (`indent-region`) applies `TAB` to every line whose first character is between point and mark.

22.5.3 Customizing Lisp Indentation

The indentation pattern for a Lisp expression can depend on the function called by the expression. For each Lisp function, you can choose among several predefined patterns of indentation, or define an arbitrary one with a Lisp program.

The standard pattern of indentation is as follows: the second line of the expression is indented under the first argument, if that is on the same line as the beginning of the expression; otherwise, the second line is indented underneath the function name. Each following line is indented under the previous line whose nesting depth is the same.

If the variable `lisp-indent-offset` is non-`nil`, it overrides the usual indentation pattern for the second line of an expression, so that such lines are always indented `lisp-indent-offset` more columns than the containing list.

The standard pattern is overridden for certain functions. Functions whose names start with `def` always indent the second line by `lisp-body-indent` extra columns beyond the open-parenthesis starting the expression.

The standard pattern can be overridden in various ways for individual functions, according to the `lisp-indent-hook` property of the function name. There are four possibilities for this property:

`nil`
: This is the same as no property; the standard indentation pattern is used.

`defun`
: The pattern used for function names that start with `def` is used for this function also.

a number, *number*
: The first *number* arguments of the function are *distinguished* arguments; the rest are considered the *body* of the expression. A line in the expression is indented according to whether the first argument on it is distinguished or not. If the argument is part of the body, the line is indented `lisp-body-indent` more columns than the open-parenthesis starting the containing expression. If the argument is distinguished and is either the first or second argument, it is indented *twice* that many extra columns. If the argument is distinguished and not the first or second argument, the standard pattern is followed for that line.

a symbol, *symbol*

symbol should be a function name; that function is called to calculate the indentation of a line within this expression. The function receives two arguments:

state The value returned by `parse-partial-sexp` (a Lisp primitive for indentation and nesting computation) when it parses up to the beginning of this line.

pos The position at which the line being indented begins.

It should return either a number, which is the number of columns of indentation for that line, or a list whose car is such a number. The difference between returning a number and returning a list is that a number says that all following lines at the same nesting level should be indented just like this one; a list says that following lines might call for different indentations. This makes a difference when the indentation is being computed by `C-M-q`; if the value is a number, `C-M-q` need not recalculate indentation for the following lines until the end of the list.

22.5.4 Customizing C Indentation

Two variables control which commands perform C indentation and when.

If `c-auto-newline` is non-`nil`, newlines are inserted both before and after braces that you insert, and after colons and semicolons. Correct C indentation is done on all the lines that are made this way.

If `c-tab-always-indent` is `nil`, the `TAB` command in C mode does indentation only if point is at the left margin or within the line's indentation. If there is non-whitespace to the left of point, then `TAB` just inserts a tab character in the buffer. Normally, this variable is `t`, and `TAB` always reindents the current line. The default behavior means that to insert a real tab character you must quote it by typing `C-q TAB`.

C does not have anything analogous to particular function names for which special forms of indentation are desirable. However, it has a different need for customization facilities: many different styles of C indentation are in common use.

There are six variables you can set to control the style that Emacs C mode uses.

`c-indent-level`

Indentation of C statements within surrounding block. The surrounding block's indentation is the indentation of the line on which the open-brace appears.

`c-continued-statement-offset`
: Extra indentation given to a substatement, such as the then-clause of an if or body of a while.

`c-brace-offset`
: Extra indentation for line if it starts with an open brace.

`c-brace-imaginary-offset`
: An open brace following other text is treated as if it were this far to the right of the start of its line.

`c-argdecl-indent`
: Indentation level of declarations of C function arguments.

`c-label-offset`
: Extra indentation for line that is a label, or case or default.

The variable `c-indent-level` controls the indentation for C statements with respect to the surrounding block. In the example

```
{
  foo ();
```

the difference in indentation between the lines is `c-indent-level`. Its standard value is 2.

If the open-brace beginning the compound statement is not at the beginning of its line, the `c-indent-level` is added to the indentation of the line, not the column of the open-brace. For example,

```
if (losing) {
  do_this ();
```

One popular indentation style is that which results from setting `c-indent-level` to 8 and putting open-braces at the end of a line in this way. I prefer to put the open-brace on a separate line.

In fact, the value of the variable `c-brace-imaginary-offset` is also added to the indentation of such a statement. Normally this variable is zero. Think of this variable as the imaginary position of the open brace, relative to the first nonblank character on the line. By setting this variable to 4 and `c-indent-level` to 0, you can get this style:

```
if (x == y) {
    do_it ();
    }
```

When `c-indent-level` is zero, the statements inside most braces will line up right under the open brace. But there is an exception made for braces in column zero, such as surrounding a function's body. The statements just inside it do not go at column zero. Instead, `c-brace-offset` and `c-continued-statement-offset` (see below) are added to produce a typical offset between brace levels, and the statements are indented that far.

`c-continued-statement-offset` controls the extra indentation for a line that starts within a statement (but not within parentheses or brackets). These lines are usually statements that are within other statements, such as the then-clauses of `if` statements and the bodies of `while` statements. This parameter is the difference in indentation between the two lines in

```
if (x == y)
  do_it ();
```

Its standard value is 2. Some popular indentation styles correspond to a value of zero for `c-continued-statement-offset`.

`c-brace-offset` is the extra indentation given to a line that starts with an open-brace. Its standard value is zero; compare

```
if (x == y)
  {
```

with

```
if (x == y)
  do_it ();
```

if `c-brace-offset` were set to 4, the first example would become

```
if (x == y)
      {
```

`c-argdecl-indent` controls the indentation of declarations of the arguments of a C function. It is absolute: argument declarations receive exactly `c-argdecl-indent` spaces. The standard value is 5, resulting in code like this:

```
char *
index (string, c)
     char *string;
     int c;
```

`c-label-offset` is the extra indentation given to a line that contains a label, a case statement, or a `default:` statement. Its standard value is -2, resulting in code like this

```
switch (c)
  {
  case 'x':
```

If `c-label-offset` were zero, the same code would be indented as

```
switch (c)
  {
    case 'x':
```

This example assumes that the other variables above also have their standard values.

I strongly recommend that you try out the indentation style produced by the standard settings of these variables, together with putting open braces on separate lines. You can see how it looks in all the C source files of GNU Emacs.

22.6 Automatic Display Of Matching Parentheses

The Emacs parenthesis-matching feature is designed to show automatically how parentheses match in the text. Whenever you type a self-inserting character that is a closing delimiter, the cursor moves momentarily to the location of the matching opening delimiter, provided that is on the screen. If it is not on the screen, some text near it is displayed in the echo area. Either way, you can tell what grouping is being closed off.

In Lisp, automatic matching applies only to parentheses. In C, it applies to braces and brackets too. Emacs knows which characters to regard as matching delimiters based on the syntax table, which is set by the major mode. See Section 31.6 [Syntax], page 311.

If the opening delimiter and closing delimiter are mismatched—such as in '`[x)`'—a warning message is displayed in the echo area. The correct matches are specified in the syntax table.

Two variables control parenthesis match display. `blink-matching-paren` turns the feature on or off; `nil` turns it off, but the default is `t` to turn match display on. `blink-matching-paren-distance` specifies how many characters back to search to find the matching opening delimiter. If the match is not found in that far, scanning stops, and nothing is displayed. This is to prevent scanning for the matching delimiter from wasting lots of time when there is no match. The default is 12,000.

When using X Windows, you can request a more powerful kind of automatic parenthesis matching by loading the `paren` library. To load it, type `M-x load-library RET paren RET`. This library turns off the usual kind of matching parenthesis display and substitutes another: whenever point is after a close parenthesis, the close parenthesis and its matching open parenthesis are both highlighted; otherwise, if point is before an open parenthesis, the matching close parenthesis is highlighted. (There is no need to highlight the open parenthesis after point because the cursor appears on top of that character.)

22.7 Manipulating Comments

The comment commands insert, kill and align comments.

`M-;` Insert or align comment (`indent-for-comment`).

`C-x ;` Set comment column (`set-comment-column`).

`C-u - C-x ;`
  Kill comment on current line (`kill-comment`).

`M-LFD` Like `RET` followed by inserting and aligning a comment (`indent-new-comment-line`).

`M-x comment-region`
  Add or remove comment delimiters on all the lines in the region.

The command that creates a comment is `M-;` (`indent-for-comment`). If there is no comment already on the line, a new comment is created, aligned at a specific column called the *comment column*. The comment is created by inserting the string Emacs thinks comments should start with (the value of `comment-start`; see below). Point is left after that string. If the text of the line extends past the comment column, then the indentation is done to a suitable boundary (usually, at least one space is inserted). If the major mode has specified a string to terminate comments, that is inserted after point, to keep the syntax valid.

`M-;` can also be used to align an existing comment. If a line already contains the string that starts comments, then `M-;` just moves point after it and re-indents it to the conventional place. Exception: comments starting in column 0 are not moved.

Some major modes have special rules for indenting certain kinds of comments in certain contexts. For example, in Lisp code, comments which start with two semicolons are indented as if they were lines of code, instead of at the comment column. Comments which start with three semicolons are supposed to start at the left margin. Emacs understands these conventions by indenting a double-semicolon comment using `TAB`, and by not changing the indentation of a triple-semicolon comment at all.

```
;; This function is just an example
;;; Here either two or three semicolons are appropriate.
(defun foo (x)
;;; And now, the first part of the function:
  ;; The following line adds one.
  (1+ x))           ; This line adds one.
```

In C code, a comment preceded on its line by nothing but whitespace is indented like a line of code.

Even when an existing comment is properly aligned, `M-;` is still useful for moving directly to the start of the comment.

`C-u - C-x ;` (`kill-comment`) kills the comment on the current line, if there is one. The indentation before the start of the comment is killed as well. If there does not appear to be a comment in the line, nothing is done. To reinsert the comment on another line, move to the end of that line, do `C-y`, and then do `M-;` to realign it. Note that `C-u - C-x ;` is not a distinct key; it is `C-x ;` (`set-comment-column`) with a negative argument. That command is programmed so that when it receives a negative argument it calls `kill-comment`. However, `kill-comment` is a valid command which you could bind directly to a key if you wanted to.

The `M-x comment-region` command adds comment delimiters to the lines that start in the region, thus commenting them out. With a negative argument, it does the opposite—it deletes comment delimiters from the lines in the region.

With a positive argument, `comment-region` adds comment delimiters and duplicates the last character of the comment start sequence as many times as the argument specifies. Thus, in Lisp mode, `C-u 2 M-x comment-region` adds '`;;`' to each line.

Duplicating the comment delimiter is a way of calling attention to the comment. It can also affect how the comment is indented. In Lisp, for proper indentation, you should use an argument of two, if between defuns, and three, if within a defun.

22.7.1 Multiple Lines of Comments

If you are typing a comment and find that you wish to continue it on another line, you can use the command `M-LFD` (`indent-new-comment-line`), which terminates the comment you are typing, creates a new blank line afterward, and begins a new comment indented under the old one. When Auto Fill mode is on, going past the fill column while typing a comment causes the comment to be continued in just this fashion. If point is not at the end of the line when `M-LFD` is typed, the text on the rest of the line becomes part of the new comment line.

22.7.2 Options Controlling Comments

The comment column is stored in the variable `comment-column`. You can set it to a number explicitly. Alternatively, the command `C-x ;` (`set-comment-column`) sets the comment column to the column point is at. `C-u C-x ;` sets the comment column to match the last comment before point in the buffer, and then does a `M-;` to align the current line's comment under the previous one. Note that `C-u - C-x ;` runs the function `kill-comment` as described above.

The variable `comment-column` is per-buffer: setting the variable in the normal fashion affects only the current buffer, but there is a default value which you can change with `setq-default`. See Section 31.2.4 [Locals], page 295. Many major modes initialize this variable for the current buffer.

The comment commands recognize comments based on the regular expression that is the value of the variable `comment-start-skip`. This regexp should not match the null string. It may match more than the comment starting delimiter in the strictest sense of the word; for example, in C mode the value of the variable is `"/\\*+ *"`, which matches extra stars and spaces after the '`/*`' itself. (Note that '`\\`' is needed in Lisp syntax to include a '`\`' in the string, which is needed to deny the first star its special meaning in regexp syntax. See Section 13.5 [Regexps], page 79.)

When a comment command makes a new comment, it inserts the value of `comment-start` to begin it. The value of `comment-end` is inserted after point, so that it will follow the text that you will insert into the comment. In C mode, `comment-start` has the value `"/* "` and `comment-end` has the value `" */"`.

The variable `comment-multi-line` controls how `M-LFD` (`indent-new-comment-line`) behaves when used inside a comment. If `comment-multi-line` is `nil`, as it normally is, then the comment on the starting line is terminated and a new comment is started on the new following line. If `comment-multi-line` is not `nil`, then the new following line is set up as part of the same comment that was found on the starting line. This is done by not inserting a terminator on the old line, and not inserting a starter on the new line. In languages where multi-line comments work, the choice of value for this variable is a matter of taste.

The variable `comment-indent-function` should contain a function that will be called to compute the indentation for a newly inserted comment or for aligning an existing comment. It is set differently by various major modes. The function is called with no arguments, but with point at the beginning of the comment, or at the end of a line if a new comment is to be inserted. It should return the column in which the comment ought to start. For example, in Lisp mode, the indent hook function bases its decision on how many semicolons begin an existing comment, and on the code in the preceding lines.

22.8 Editing Without Unbalanced Parentheses

`M-(`
: Put parentheses around next sexp(s) (`insert-parentheses`).

`M-)`
: Move past next close parenthesis and re-indent (`move-over-close-and-reindent`).

The commands `M-(` (`insert-parentheses`) and `M-)` (`move-over-close-and-reindent`) are designed to facilitate a style of editing which keeps parentheses balanced at all times. `M-(` inserts a pair of parentheses, either together as in '`()`', or, if given an argument, around the next several sexps, and leaves point after the open parenthesis. Instead of typing `( F O O )`, you can type `M-( F O O`, which has the same effect except for leaving the cursor before the close parenthesis. Then you can type `M-)`, which moves past the close parenthesis, deleting any indentation preceding it (in this example there is none), and indenting with `LFD` after it.

`M-(` may insert a space before the open parenthesis, depending on the syntax class or the preceding character. Set `parens-dont-require-spaces` to a non-`nil` value if you wish to inhibit this.

22.9 Completion for Symbol Names

Usually completion happens in the minibuffer. But one kind of completion is available in all buffers: completion for symbol names.

The character `M-TAB` runs a command to complete the partial symbol before point against the set of meaningful symbol names. Any additional characters determined by the partial name are inserted at point.

If the partial name in the buffer has more than one possible completion and they have no additional characters in common, a list of all possible completions is displayed in another window.

There are two ways of determining the set of legitimate symbol names to complete against. In most major modes, this uses a tag table (see Section 22.12 [Tags], page 179); the legitimate symbol names are the tag names listed in the tag table file. The command which implements this is `complete-tag`.

In Emacs-Lisp mode, the name space for completion normally consists of nontrivial symbols present in Emacs—those that have function definitions, values or properties. However, if there is an open-parenthesis immediately before the beginning of the partial symbol, only symbols with function definitions are considered as completions. The command which implements this is `lisp-complete-symbol`.

22.10 Documentation Commands

As you edit Lisp code to be run in Emacs, the commands `C-h f` (`describe-function`) and `C-h v` (`describe-variable`) can be used to print documentation of functions and variables that you want to call. These commands use the minibuffer to read the name of a function or variable to document, and display the documentation in a window.

For extra convenience, these commands provide default arguments based on the code in the neighborhood of point. `C-h f` sets the default to the function called in the innermost list containing point. `C-h v` uses the symbol name around or adjacent to point as its default.

Documentation on Unix commands, system calls and libraries can be obtained with the `M-x manual-entry` command. This reads a topic as an argument, and displays the text on that topic from the Unix manual. `manual-`

`entry` starts a background process that formats the manual page, by running the `man` program. The result goes in a buffer named '`*man` *topic*`*`'. These buffers have a special major mode that facilitates scrolling and examining other manual pages.

Eventually the GNU project hopes to replace most man pages with better-organized manuals that you can browse with Info. See Section 8.5 [Misc Help], page 49. Since this process is only partially completed, it is still useful to read manual pages.

22.11 Change Logs

The Emacs command `C-x 4 a` adds a new entry to the change log file for the file you are editing (`add-change-log-entry-other-window`).

A change log file contains a chronological record of when and why you have changed a program, consisting of a sequence of entries describing individual changes. Normally it is kept in a file called '`ChangeLog`' in the same directory as the file you are editing, or one of its parent directories. A single '`ChangeLog`' file can record changes for all the files in its directory and all its subdirectories.

A change log entry starts with a header line that contains your name and the current date. Aside from these header lines, every line in the change log starts with a space or a tab. The bulk of the entry consists of *items*, each of which starts with a line starting with whitespace and a star. Here are two entries, each with two items:

```
Wed May  5 14:11:45 1993  Richard Stallman  (rms@mole.gnu.ai.mit.edu)

* man.el: Rename functions and variables ‘man-*’ to ‘Man-*’.
(manual-entry): Make prompt string clearer.

* simple.el (blink-matching-paren-distance): Change default to 12,000.

Tue May  4 12:42:19 1993  Richard Stallman  (rms@mole.gnu.ai.mit.edu)

* vc.el (minor-mode-map-alist): Don’t use it if it’s void.
(vc-cancel-version): Doc fix.
```

One entry can describe several changes; each change should have its own item. Normally there should be a blank line between items. When items are related (parts of the same change, in different places), group them by leaving no blank line between them. The second entry above contains two items grouped in this way.

`C-x 4 a` visits the change log file and creates a new entry unless the most recent entry is for today's date and your name. It also creates a new item for the current file. For many languages, it can even guess the name of the function or other object that was changed.

The change log file is visited in Change Log mode. Each bunch of grouped items counts as one paragraph, and each entry is considered a page. This facilitates editing the entries. `LFD` and auto-fill indent each new line like the previous line; this is convenient for entering the contents of an entry.

22.12 Tag Tables

A *tag table* is a description of how a multi-file program is broken up into files. It lists the names of the component files and the names and positions of the functions (or other named subunits) in each file. Grouping the related files makes it possible to search or replace through all the files with one command. Recording the function names and positions makes possible the `M-.` command which you can use to find the definition of a function without having to know which of the files it is in.

Tag tables are stored in files called *tag table files*. The conventional name for a tag table file is '`TAGS`'.

Each entry in the tag table records the name of one tag, the name of the file that the tag is defined in (implicitly), and the position in that file of the tag's definition.

Just what names from the described files are recorded in the tag table depends on the programming language of the described file. They normally include all functions and subroutines, and may also include global variables, data types, and anything else convenient. Each name recorded is called a *tag*.

22.12.1 Source File Tag Syntax

In Lisp code, any function defined with `defun`, any variable defined with `defvar` or `defconst`, and in general the first argument of any expression that starts with '`(def`' in column zero, is a tag.

In Scheme code, tags include anything defined with `def` or with a construct whose name starts with '`def`'. They also include variables set with `set!` at top level in the file.

In C code, any C function is a tag, and so is any typedef if `-t` is specified when the tag table is constructed.

In Yacc or Bison input files, each rule defines as a tag the nonterminal it constructs. The portions of the file that contain C code are parsed as C code.

In Fortran code, functions and subroutines are tags.

In Prolog code, a tag name appears at the left margin.

In assembler code, labels appearing at the beginning of a line, followed by a colon, are tags.

In LaTeX text, the argument of any of the commands `\chapter`, `\section`, `\subsection`, `\subsubsection`, `\eqno`, `\label`, `\ref`, `\cite`, `\bibitem` and `\typeout` is a tag.

22.12.2 Creating Tag Tables

The `etags` program is used to create a tag table file. It knows the syntax of several languages, as described in the previous section. Here is how to run `etags`:

```
etags inputfiles...
```

The `etags` program reads the specified files, and writes a tag table named '`TAGS`' in the current working directory. `etags` recognizes the language used in an input file based on its file name and contents; there are no switches for specifying the language. The `-t` switch tells `etags` to record typedefs in C code as tags.

If the tag table data become outdated due to changes in the files described in the table, the way to update the tag table is the same way it was made in the first place. It is not necessary to do this often.

If the tag table fails to record a tag, or records it for the wrong file, then Emacs cannot possibly find its definition. However, if the position recorded in the tag table becomes a little bit wrong (due to some editing in the file that the tag definition is in), the only consequence is a slight delay in finding the tag. Even if the stored position is very wrong, Emacs will still find the tag, but it must search the entire file for it.

So you should update a tag table when you define new tags that you want to have listed, or when you move tag definitions from one file to another, or when changes become substantial. Normally there is no need to update the tag table after each edit, or even every day.

22.12.3 Selecting a Tag Table

Emacs has at any time one *selected* tag table, and all the commands for working with tag tables use the selected one. To select a tag table, type `M-x visit-tags-table`, which reads the tag table file name as an argument. The name '`TAGS`' in the default directory is used as the default file name.

All this command does is store the file name in the variable `tags-file-name`. Emacs does not actually read in the tag table contents until you try to use them. Setting this variable yourself is just as good as using `visit-tags-table`. The variable's initial value is `nil`; that value tells all the commands for working with tag tables that they must ask for a tag table file name to use.

Using `visit-tags-table` to load a new tag table does not discard the other tables previously loaded. The other tags commands use all the tag tables that are loaded; the first one they use is the one that mentions the current visited file.

You can specify a precise list of tag tables by setting the variable `tags-table-list` to a list of strings, like this:

```
(setq tags-table-list
      '("~/emacs" "/usr/local/lib/emacs/src"))
```

This tells the tags commands to look at the '`TAGS`' files in your '`~/emacs`' directory and in the '`/usr/local/lib/emacs/src`' directory. The order depends on which file you are in and which tags table mentions that file, as explained above.

22.12.4 Finding a Tag

The most important thing that a tag table enables you to do is to find the definition of a specific tag.

`M-.` *tag* `RET`
: Find first definition of *tag* (`find-tag`).

`C-u M-.`
: Find next alternate definition of last tag specified.

`C-u - M-.`
: Go back to previous tag found.

`M-x find-tag-regexp RET` *pattern* `RET`
: Find a tag whose name matches *pattern*.

`C-u M-x find-tag-regexp`
: Find the next tag whose name matches the last pattern used.

`C-x 4 .` *tag* `RET`
: Find first definition of *tag*, but display it in another window (`find-tag-other-window`).

`C-x 5 .` *tag* `RET`
: Find first definition of *tag*, and create a new frame to select the buffer (`find-tag-other-frame`).

`M-.` (`find-tag`) is the command to find the definition of a specified tag. It searches through the tag table for that tag, as a string, and then uses the tag table info to determine the file that the definition is in and the approximate character position in the file of the definition. Then `find-tag` visits that file, moves point to the approximate character position, and searches ever-increasing distances away to find the tag definition.

If an empty argument is given (just type `RET`), the sexp in the buffer before or around point is used as the *tag* argument. See Section 22.2 [Lists], page 164, for info on sexps.

You don't need to give `M-.` the full name of the tag; a part will do. This is because `M-.` finds tags in the table which contain *tag* as a substring. However, it prefers an exact match to a substring match.

To find other tags that match the same substring, give `find-tag` a numeric argument, as in `C-u M-.`; this does not read a tag name, but continues searching the tag table's text for another tag containing the same substring last used. If you have a real META key, `M-0 M-.` is an easier alternative to `C-u M-.`.

Like most commands that can switch buffers, `find-tag` has a variant that displays the new buffer in another window, and one that makes a new frame for it. The former is `C-x 4 .`, which invokes the command `find-tag-other-window`. The latter is `C-x 5 .`, which invokes `find-tag-other-frame`.

To move back to places you've found tags recently, use `C-u - M-.`; more generally, `M-.` with a negative numeric argument. This command can take you to another buffer. `C-x 4 .` with a negative argument finds the previous tag location in another window.

The new command `M-x find-tag-regexp` visits the tags that match a specified regular expression. It is just like `M-.` except that it does regexp matching instead of substring matching.

Emacs comes with a tag table file '`src/TAGS`' that includes all the Lisp libraries and all the C sources of Emacs. By specifying this file with `visit-tags-table` and then using `M-.` you can quickly find the source for any Emacs function.

22.12.5 Searching and Replacing with Tag Tables

The commands in this section visit and search all the files listed in the selected tag table, one by one. For these commands, the tag table serves only to specify a sequence of files to search. A related command is `M-x grep` (see Section 23.1 [Compilation], page 199).

`M-x tags-search`
: Search for the specified regexp through the files in the selected tag table.

`M-x tags-query-replace`
: Perform a `query-replace` on each file in the selected tag table.

`M-,`
: Restart one of the commands above, from the current location of point (`tags-loop-continue`).

`M-x tags-search` reads a regexp using the minibuffer, then searches for matches in all the files in the selected tag table, one file at a time. It displays the name of the file being searched so you can follow its progress. As soon as it finds an occurrence, `tags-search` returns.

Having found one match, you probably want to find all the rest. To find one more match, type `M-,` (`tags-loop-continue`) to resume the `tags-search`. This searches the rest of the current buffer, followed by the remaining files of the tag table.

`M-x tags-query-replace` performs a single `query-replace` through all the files in the tag table. It reads a regexp to search for and a string to replace with, just like ordinary `M-x query-replace-regexp`. It searches much like `M-x tags-search` but repeatedly, processing matches according to your input. See Section 13.7 [Replace], page 84, for more information on query replace.

It is possible to get through all the files in the tag table with a single invocation of `M-x tags-query-replace`. But since any unrecognized character causes the command to exit, you may need to continue where you left off. `M-,` can be used for this. It resumes the last tags search or replace command that you did.

The commands in this section carry out much broader searches than the `find-tags` family. The `find-tags` commands search only for definitions of tags that match your substring or regexp. The commands `tags-search` and `tags-query-replace` find every occurrence of the regexp, as ordinary search commands and replace commands do in the current buffer.

These commands create buffers only temporarily for the files that they have to search (those which are not already visited in Emacs buffers). Buffers in which no match is found are quickly killed; the others continue to exist.

It may have struck you that `tags-search` is a lot like `grep`. You can also run `grep` itself as an inferior of Emacs and have Emacs show you the matching lines one by one. This works mostly the same as running a compilation and having Emacs show you where the errors were. See Section 23.1 [Compilation], page 199.

22.12.6 Stepping Through a Tag Table

If you wish to process all the files in the selected tag table, but not in the specific ways that `M-x tags-search` and `M-x tags-query-replace` do, you can use `M-x next-file` to visit the files one by one.

`C-u M-x next-file`
: Visit the first file in the tag table, and prepare to advance sequentially by files.

`M-x next-file`
: Visit the next file in the selected tag table.

22.12.7 Tag Table Inquiries

`M-x list-tags`
: Display a list of the tags defined in a specific program file.

`M-x tags-apropos`
: Display a list of all tags matching a specified regexp.

`M-x list-tags` reads the name of one of the files described by the selected tag table, and displays a list of all the tags defined in that file. The "file name" argument is really just a string to compare against the names recorded in the tag table; it is read as a string rather than as a file name. Therefore, completion and defaulting are not available, and you must enter the string the same way it appears in the tag table. Do not include a directory as part of the file name unless the file name recorded in the tag table includes a directory.

`M-x tags-apropos` is like `apropos` for tags. It reads a regexp, then finds all the tags in the selected tag table whose entries match that regexp, and displays the tag names found.

You can also perform completion in the buffer on the name space of tag names in the current tag tables. See Section 22.9 [Symbol Completion], page 177.

22.13 Merging Files with Emerge

It's not unusual for programmers to get their signals crossed and modify the same program in two different directions. To recover from this confusion, you need to merge the two versions. Emerge makes this easier. See also Section 15.9 [Comparing Files], page 116.

22.13.1 Overview of Emerge

To start Emerge, run one of these four commands:

`M-x emerge-files`
: Merge two specified files.

`M-x emerge-files-with-ancestor`
: Merge two specified files, with reference to a common ancestor.

`M-x emerge-buffers`
: Merge two buffers.

`M-x emerge-buffers-with-ancestor`
: Merge two buffers with reference to a common ancestor in a third buffer.

The Emerge commands compare two files or buffers, and display the comparison in three buffers: one for each input text (the *A buffer* and the *B buffer*), and one (the *merge buffer*) where merging takes place. The merge buffer shows the full merged text, not just the differences. Wherever the two input texts differ, you can choose which one of them to include in the merge buffer.

The Emerge commands that take input from existing buffers use only the accessible portions of those buffers, if they are narrowed (see Section 30.4 [Narrowing], page 282).

If a common ancestor version is available, from which the two texts to be merged were both derived, Emerge can use it to guess which alternative is right. Wherever one current version agrees with the ancestor, Emerge presumes that the other current version is a deliberate change which should be kept in the merged version. Use the '`with-ancestor`' commands if you want to specify a common ancestor text. These commands read three file or buffer names—variant A, variant B, and the common ancestor.

After the comparison is done and the buffers are prepared, the interactive merging starts. You control the merging by typing special commands in the merge buffer. The merge buffer shows you a full merged text, not just differences. For each run of differences between the input texts, you can choose which one of them to keep, or edit them both together.

The merge buffer uses a special major mode, Emerge mode, with commands for making these choices. But you can also edit the buffer with ordinary Emacs commands.

At any given time, the attention of Emerge is focused on one particular difference, called the *selected* difference. This difference is marked off in the three buffers like this:

```
vvvvvvvvvvvvvvvvvvvv
text that differs
^^^^^^^^^^^^^^^^^^^^
```

Emerge numbers all the differences sequentially and the mode line always shows the number of the selected difference.

Normally, the merge buffer starts out with the A version of the text. But when the A version of a part of the buffer agrees with the common ancestor, then the B version is preferred for that part.

Emerge leaves the merged text in the merge buffer when you exit. At that point, you can save it in a file with `C-x C-w`. If you give a prefix argument to `emerge-files` or `emerge-files-with-ancestor`, it reads the name of the output file using the minibuffer. (This is the last file name those commands read.) Then exiting from Emerge saves the merged text in the output file.

If you abort Emerge with `C-]`, the output is not saved.

22.13.2 Submodes of Emerge

You can choose between two modes for giving merge commands: Fast mode and Edit mode. In Fast mode, basic Emerge commands are single characters, but ordinary Emacs commands are disabled. This is convenient if you use only Emerge commands.

In Edit mode, all Emerge commands start with the prefix key `C-c C-c`, and the normal Emacs commands are also available. This allows editing the merge buffer, but slows down Emerge operations.

Use `e` to switch to Edit mode, and `C-c C-c f` to switch to Fast mode. The mode line indicates Edit and Fast modes with '`E`' and '`F`'.

Emerge has two additional submodes that affect how particular merge commands work: Auto Advance mode and Skip Prefers mode.

If Auto Advance mode is in effect, the `a` and `b` commands advance to the next difference. This lets you go through the merge faster as long as you simply choose one of the alternatives from the input. The mode line indicates Auto Advance mode with '`A`'.

If Skip Prefers mode is in effect, the `n` and `p` commands skip over differences in states prefer-A and prefer-B. Thus you see only differences for which neither version is presumed "correct". The mode line indicates Skip Prefers mode with '`S`'.

Use the command `s a` (`emerge-auto-advance-mode`) to set or clear Auto Advance mode. Use `s s` (`emerge-skip-prefers-mode`) to set or clear Skip Prefers mode. These commands turn on the mode with a positive argument, turns it off with a negative or zero argument, and toggle the mode with no argument.

22.13.3 State of a Difference

In the merge buffer, a difference is marked with lines of '`v`' and '`^`' characters. Each difference has one of these seven states:

A
: The difference is showing the A version. The `a` command always produces this state; the mode line indicates it with '`A`'.

B
: The difference is showing the B version. The `b` command always produces this state; the mode line indicates it with '`B`'.

default-A
default-B
: The difference is showing the A or the B state by default, because you haven't made a choice. All differences start in the default-A state (and thus the merge buffer is a copy of the A buffer), except those for which one alternative is "preferred" (see below).

  When you select a difference, its state changes from default-A or default-B to plain A or B. Thus, the selected difference never has state default-A or default-B, and these states are never displayed in the mode line.

The command `d a` chooses default-A as the default state, and `d b` chooses default-B. This chosen default applies to all differences which you haven't selected and for which no alternative is preferred. If you are moving through the merge sequentially, the differences you haven't selected are those following the selected one. Thus, while moving sequentially, you can effectively make the A version the default for some sections of the merge buffer and the B version the default for others by using `d a` and `d b` at the end of each section.

prefer-A
prefer-B
: The difference is showing the A or B state because it is *preferred*. This means that you haven't made an explicit choice, but one alternative seems likely to be right because the other alternative agrees with the common ancestor. Thus, where the A buffer agrees with the common ancestor, the B version is preferred, because chances are it is the one that was actually changed.

These two states are displayed in the mode line as '`A*`' and '`B*`'.

combined
: The difference is showing a combination of the A and B states, as a result of the `x c` or `x C` commands.

Once a difference is in this state, the `a` and `b` commands don't do anything to it unless you give them a prefix argument.

The mode line displays this state as '`comb`'.

22.13.4 Merge Commands

Here are the Merge commands for Fast mode; in Edit mode, precede them with `C-c C-c`:

`p` Select the previous difference.

`n` Select the next difference.

`a` Choose the A version of this difference.

`b` Choose the B version of this difference.

`j` Select a particular difference; specify the sequence number of that difference as a prefix argument.

`.` Select the difference containing point. You can use this command in the merge buffer or in the A or B buffer.

`q` Quit—finish the merge.

`C-]` Abort—exit merging and do not save the output.

`f` Go into Fast mode. (In Edit mode, this is actually `C-c C-c f`.)

`e` Go into Edit mode.

`l` Recenter (like `C-l`) all three windows.

`-` Specify part of a prefix numeric argument.

digit Also specify part of a prefix numeric argument.

`d a` Choose the A version as the default from here down in the merge buffer.

`d b` Choose the B version as the default from here down in the merge buffer.

`c a` Copy the A version of this difference into the kill ring.

`c b` Copy the B version of this difference into the kill ring.

`i a` Insert the A version of this difference at the point.

`i b` Insert the B version of this difference at the point.

`m` Put the point and mark around the difference region.

`^` Scroll all three windows down (like `M-v`).

`v` Scroll all three windows up (like `C-v`).

`<` Scroll all three windows left (like `C-x <`).

`>` Scroll all three windows right (like `C-x >`).

`|` Reset horizontal scroll on all three windows.

`x 1` Shrink the merge window to one line. (Use `C-u l` to restore it to full size.)

`x c` Combine the two versions of this difference.

`x f` Show the files/buffers Emerge is operating on in Help window. (Use `C-u l` to restore windows.)

`x j` Join this difference with the following one. (`C-u x j` joins this difference with the previous one.)

`x s` Split this difference into two differences. Before you use this command, position point in each of the three buffers to the place where you want to split the difference.

`x t` Trim identical lines off top and bottom of the difference. Such lines occur when the A and B versions are identical but differ from the ancestor version.

22.13.5 Exiting Emerge

The `q` command (`emerge-quit`) finishes the merge, storing the results into the output file if you specified one. It restores the A and B buffers to their proper contents, or kills them if they were created by Emerge and you haven't changed them. It also disables the Emerge commands in the merge buffer, since executing them later could damage the contents of the various buffers.

`C-]` aborts the merge. This means exiting without writing the output file. If you didn't specify an output file, then there is no real difference between aborting and finishing the merge.

If Emerge was called from another Lisp program, then its return value is `t` for successful completion, or `nil` if you abort.

22.13.6 Combining the Two Versions

Sometimes you want to keep *both* alternatives for a particular locus. To do this, use `x c`, which edits the merge buffer like this:

```
#ifdef NEW
version from A file
#else /* NEW */
version from B file
#endif /* NEW */
```

While this example shows C preprocessor conditionals delimiting the two alternative versions, you can specify the strings you want by setting the variable `emerge-combine-versions-template` to a string of your choice. In the string, '`%a`' says where to put version A, and '`%b`' says where to put version B. The default setting, which produces the results shown above, looks like this:

```
"#ifdef NEW\n%a#else /* NEW */\n%b#endif /* NEW */\n"
```

22.13.7 Fine Points of Emerge

During the merge, you mustn't try to edit the A and B buffers yourself. Emerge modifies them temporarily, but ultimately puts them back the way they were.

You can have any number of merges going at once—just don't use any one buffer as input to more than one merge at once, since the temporary changes made in these buffers would get in each other's way.

Starting Emerge can take a long time because it needs to compare the files fully. Emacs can't do anything else until `diff` finishes. Perhaps in the future someone will change Emerge to do the comparison in the background when the input files are large—then you could keep on doing other things with Emacs until Emerge gets ready to accept commands.

After setting up the merge, Emerge runs the hook `emerge-startup-hook` (see Section 31.2.3 [Hooks], page 294).

22.14 C Mode

In addition to the facilities of typical programming language major modes (see Section 22.1 [Program Modes], page 163), C mode has various special facilities.

`M-a`
`M-e`
: In C mode, `M-a` and `M-e` now move by complete C statements (`c-beginning-of-statement` and `c-end-of-statement`). These commands do ordinary, textual sentence motion when in or next to a comment.

`M-q`
: `M-q` in C mode runs `c-fill-paragraph`, which is designed for filling C comments. (We assume you don't want to fill the actual C code in a C program.)

`C-c C-u`
: Move back to the containing preprocessor conditional, setting the mark at the starting point (`c-up-conditional`).

A prefix argument acts as a repeat count. With a negative argument, this command moves forward to the end of the containing preprocessor conditional. When going backwards, '`#elif`' acts like '`#else`' followed by '`#if`'. When going forwards, '`#elif`' is ignored.

`C-c C-n`
: Move forward across the next preprocessor conditional, setting the mark at the starting point (`c-forward-conditional`).

`C-c C-p`
: Move backward across the previous preprocessor conditional, setting the at the starting point (`c-backward-conditional`).

`M-x c-macro-expand`
: When you are debugging C code that uses macros, sometimes it is hard to figure out precisely how the macros expand. The command `M-x c-macro-expand` runs the C preprocessor and shows you what expansion results from the region. The portion of the buffer before the region is also included in preprocessing, for the sake of macros defined there, but the output from this part isn't shown.

`M-x c-backslash-region`
: Insert or align '`\`' characters at the ends of the lines of the region, except for the last such line. This is useful after writing or editing a C macro definition.

If a line already ends in '\', this command adjusts the amount of whitespace before it. Otherwise, it inserts a new '\'.

C++ mode is like C mode, except that it understands C++ comment syntax and certain other differences between C and C++. It also has a command `M-x fill-c++-comment`, which fills a paragraph made of C++ comment lines.

The command `comment-region` is useful in C++ mode for commenting out several consecutive lines, or removing the commenting out of such lines. (You don't need this command with C comment syntax because you don't need to put comment delimiters on each line.) See Section 22.7.2 [Comments], page 175.

22.15 Fortran Mode

Fortran mode provides special motion commands for Fortran statements and subprograms, and indentation commands that understand Fortran conventions of nesting, line numbers and continuation statements. Fortran mode has it's own Auto Fill mode that breaks long lines into proper Fortran continuation lines.

Special commands for comments are provided because Fortran comments are unlike those of other languages.

Built-in abbrevs optionally save typing when you insert Fortran keywords.

Use `M-x fortran-mode` to switch to this major mode. This command runs the hook `fortran-mode-hook` (see Section 31.2.3 [Hooks], page 294).

Fortran mode was contributed by Michael Prange. It has been updated by Stephen A. Wood who has collated the contributions and suggestions of many users.

22.15.1 Motion Commands

Fortran mode provides special commands to move by subprograms (functions and subroutines) and by statements. There is also a command to put the region around one subprogram, convenient for killing it or moving it.

`C-M-a` Move to beginning of subprogram (`beginning-of-fortran-subprogram`).

`C-M-e` Move to end of subprogram (`end-of-fortran-subprogram`).

`C-M-h` Put point at beginning of subprogram and mark at end (`mark-fortran-subprogram`).

`C-c C-n` Move to beginning of current or next statement (`fortran-next-statement`).

`C-c C-p` Move to beginning of current or previous statement (`fortran-previous-statement`).

22.15.2 Fortran Indentation

Special commands and features are needed for indenting Fortran code in order to make sure various syntactic entities (line numbers, comment line indicators and continuation line flags) appear in the columns that are required for standard Fortran.

22.15.2.1 Fortran Indentation Commands

`TAB` Indent the current line (`fortran-indent-line`).

`LFD` Indent the current and start a new indented line (`fortran-indent-new-line`).

`M-LFD` Break the current line and set up a continuation line.

`C-M-q` Indent all the lines of the subprogram point is in (`fortran-indent-subprogram`).

Fortran mode redefines `TAB` to reindent the current line for Fortran (`fortran-indent-line`). This command indents Line numbers and continuation markers to their required columns, and independently indents the body of the statement based on its nesting in the program.

The key `LFD` runs the command `fortran-indent-new-line`, which reindents the current line then makes and indents a new line. This command is useful to reindent the closing statement of '`do`' loops and other blocks before starting a new line.

The key `C-M-q` runs `fortran-indent-subprogram`, a command to reindent all the lines of the Fortran subprogram (function or subroutine) containing point.

The key `M-LFD` runs `fortran-split-line`, which splits a line in the appropriate fashion for Fortran. In a non-comment line, the second half becomes a continuation line and is indented accordingly. In a comment line, both halves become separate comment lines.

22.15.2.2 Continuation Lines

Most modern Fortran compilers allow two ways of writing continuation lines. If the first non-space character on a line is in column 5, then that line is a continuation of the previous line. We call this *fixed format*. (In GNU Emacs we always count columns from 0.) A line that starts with a tab character followed by any digit except '0' is also a continuation line. We call this style of continuation *tab format*.

Fortran mode can make either style of continuation line, but you must specify which one you prefer. The value of the variable `indent-tabs-mode` controls the choice: `nil` for fixed format, and non-`nil` for tab format. You can tell which style is presently in effect by the presence or absence of the string '`Tab`' in the mode line.

If the text on a line starts with the conventional Fortran continuation marker '`$`', or if it begins with any non-whitespace character in column 5, Fortran mode treats it as a continuation line. When you indent a continuation line with `TAB`, it converts the line to the current continuation style. When you split a Fortran statement with `M-LFD`, the continuation marker on the newline is created according to the continuation style.

The setting of continuation style affects several other aspects of editing in Fortran mode. In fixed format mode, the minimum column number for the body of a statement is 6. Lines inside of Fortran blocks that are indented to larger column numbers always use only the space character for whitespace. In tab format mode, the minimum column number for the statement body is 8, and the whitespace before column 8 must always consist of one tab character.

When you enter Fortran mode for an existing file, it tries to deduce the proper continuation style automatically from the file contents. The first line that begins with either a tab character or six spaces determines the choice. The variable `fortran-analyze-depth` specifies how many lines to consider (at the beginning of the file); if none of those lines indicates a style, then the variable `fortran-tab-mode-default` specifies the style. If it is `nil`, that specifies fixed format, and non-`nil` specifies tab format.

22.15.2.3 Line Numbers

If a number is the first non-whitespace in the line, Fortran indentation assumes it is a line number and moves it to columns 0 through 4. (Columns always count from 0 in GNU Emacs.)

Line numbers of four digits or less are normally indented one space. The variable `fortran-line-number-indent` controls this; it specifies the maximum indentation a line number can have. Line numbers are indented to right-justify them to end in column 4 unless that would require more than this maximum indentation. The default value of the variable is 1.

Simply inserting a line number is enough to indent it according to these rules. As each digit is inserted, the indentation is recomputed. To turn off this feature, set the variable `fortran-electric-line-number` to `nil`. Then inserting line numbers is like inserting anything else.

22.15.2.4 Syntactic Conventions

Fortran mode assumes that you follow certain conventions that simplify the task of understanding a Fortran program well enough to indent it properly:

- Two nested `do` loops never share a `continue` statement.
- Fortran keywords such as `if`, `else`, `then`, `do` and others are written without embedded whitespace or line breaks.

  Fortran compilers generally ignore whitespace outside of string constants, but Fortran mode does not recognize these keywords if they are not contiguous. Constructs such as `else if` or `end do` are acceptable, but the second word should be on the same line as the first and not on a continuation line.

If you fail to follow these conventions, the indentation commands may indent some lines unaesthetically. However, a correct Fortran program retains its meaning when reindented even if the conventions are not followed.

22.15.2.5 Variables for Fortran Indentation

Several additional variables control how Fortran indentation works:

`fortran-do-indent`
: Extra indentation within each level of `do` statement (default 3).

`fortran-if-indent`
: Extra indentation within each level of `if` statement (default 3). This value is also used for extra indentation within each level of the Fortran 90 `where` statement.

`fortran-structure-indent`
: Extra indentation within each level of `structure`, `union`, or `map` statements (default 3).

`fortran-continuation-indent`
: Extra indentation for bodies of continuation lines (default 5).

`fortran-check-all-num-for-matching-do`
: If this is `nil`, indentation assumes that each `do` statement ends on a `continue` statement. Therefore, when computing indentation for a statement other than `continue`, it can save time by not checking for a `do` statement ending there. If this is non-`nil`, indenting any numbered statement must check for a `do` that ends there. The default is `nil`.

`fortran-blink-matching-if`
: If this is `t`, indenting an '`endif`' statement moves the cursor momentarily to the matching '`if`' statement to show where it is. The default is `nil`.

`fortran-minimum-statement-indent-fixed`
: Minimum indentation for fortran statements when using fixed format continuation line style. Statement bodies are never indented less than this much. The default is 6.

`fortran-minimum-statement-indent-tab`
: Minimum indentation for fortran statements for tab format continuation line style. Statement bodies are never indented less than this much. The default is 8.

22.15.3 Comments

The usual Emacs comment commands assume that a comment can follow a line of code. In Fortran, the standard comment syntax requires an entire line to be just a comment. Therefore, Fortran mode replaces the standard Emacs comment commands and defines some new variables.

Fortran mode can also handle a nonstandard comment syntax where comments start with '`!`' and can follow other text. Because only some Fortran compilers accept this syntax, Fortran mode will not insert such comments unless you have said in advance to do so. To do this, set the variable `comment-start` to '`"!"`' (see Section 31.2 [Variables], page 292).

`M-;`
: Align comment or insert new comment (`fortran-comment-indent`).

`C-x ;`
: Applies to nonstandard '`!`' comments only.

`C-c ;`
: Turn all lines of the region into comments, or (with argument) turn them back into real code (`fortran-comment-region`).

`M-;` in Fortran mode is redefined as the command `fortran-comment-indent`. Like the usual `M-;` command, this recognizes any kind of existing comment and aligns its text appropriately; if there is no existing comment, a comment is inserted and aligned. But inserting and aligning comments are not the same in Fortran mode as in other modes.

When a new comment must be inserted, if the current line is blank, a full-line comment is inserted. On a non-blank line, a nonstandard '`!`' comment is inserted if you have said you want to use them. Otherwise a full-line comment is inserted on a new line before the current line.

Nonstandard '`!`' comments are aligned like comments in other languages, but full-line comments are different. In a standard full-line comment, the comment delimiter itself must always appear in column zero. What can be aligned is the text within the comment. You can choose from three styles of alignment by setting the variable `fortran-comment-indent-style` to one of these values:

`fixed`
: Align the text at a fixed column, which is the sum of `fortran-comment-line-extra-indent` and the minimum statement indentation. This is the default.

  The minimum statement indentation is `fortran-minimum-statement-indent-fixed` for fixed format continuation line style and `fortran-minimum-statement-indent-tab` for tab format style.

`relative`
: Align the text as if it were a line of code, but with an additional `fortran-comment-line-extra-indent` columns of indentation.

`nil`
: Don't move text in full-line columns automatically at all.

In addition, you can specify the character to be used to indent within full-line comments by setting the variable `fortran-comment-indent-char` to the single-character string you want to use.

Fortran mode introduces two variables `comment-line-start` and `comment-line-start-skip` which play for full-line comments the same roles played by `comment-start` and `comment-start-skip` for ordinary text-following comments. Normally these are set properly by Fortran mode so you do not need to change them.

The normal Emacs comment command `C-x ;` has not been redefined. If you use '`!`' comments, this command can be used with them. Otherwise it is useless in Fortran mode.

The command `C-c ;` (`fortran-comment-region`) turns all the lines of the region into comments by inserting the string '`C$$$`' at the front of each one. With a numeric argument, it turns the region back into live code by deleting '`C$$$`' from the front of each line in it. The string used for these comments can be controlled by setting the variable `fortran-comment-region`. Note that here we have an example of a command and a variable with the same name; these two uses of the name never conflict because in Lisp and in Emacs it is always clear from the context which one is meant.

22.15.4 Fortran Auto Fill Mode

Fortran Auto Fill mode is a minor mode which automatically splits Fortran statements as you insert them when they become too wide. Splitting a statement involves making continuation lines. This splitting happens when you type `SPC`, `RET`, or `TAB`, and also in the Fortran indentation commands.

`M-x fortran-auto-fill-mode` turns Fortran Auto Fill mode on if it was off, or off if it was on. This command works the same as `M-x auto-fill-mode` does for normal Auto Fill mode (see Section 21.5 [Filling], page 147). A positive numeric argument turns Fortran Auto Fill mode on, and a negative argument turns it off. You can see when Fortran Auto Fill mode is in effect by the presence of the word '`Fill`' in the mode line, inside the parentheses. Fortran Auto Fill mode is a minor mode, turned on or off for each buffer individually. See Section 31.1 [Minor Modes], page 291.

Fortran Auto Fill mode breaks lines at spaces or delimiters when the lines get longer than the desired width (the value of `fill-column`). The delimiters that Fortran Auto Fill mode may break at are '`,`', '`'`', '`+`', '`-`', '`/`', '`*`', '`=`', and '`)`'. The line break comes after the delimiter if the variable `fortran-break-before-delimiters` is `nil`. Otherwise (and by default), the break comes before the delimiter.

By default, Fortran Auto Fill mode is not enabled. If you want this feature turned on permanently, add a hook function to `fortran-mode-hook` to execute `(fortran-auto-fill-mode 1)`. See Section 31.2.3 [Hooks], page 294.

22.15.5 Columns

`C-c C-r` Display a "column ruler" momentarily above the current line (`fortran-column-ruler`).

`C-c C-w` Split the current window horizontally temporarily so that it is 72 columns wide. This may help you avoid making lines longer than the 72 character limit that some fortran compilers impose (`fortran-window-create-momentarily`).

The command `C-c C-r` (`fortran-column-ruler`) shows a column ruler momentarily above the current line. The comment ruler is two lines of text that show you the locations of columns with special significance in Fortran programs. Square brackets show the limits of the columns for line numbers, and curly brackets show the limits of the columns for the statement body. Column numbers appear above them.

Note that the column numbers count from zero, as always in GNU Emacs. As a result, the numbers may be one less than those you are familiar with; but the positions they indicate in the line are standard for Fortran.

The text used to display the column ruler is the value of the variable `fortran-comment-ruler`. By changing this variable, you can change the display.

For even more help, use `C-c C-w` (`fortran-window-create`), a command which splits the current window horizontally, making a window 72 columns wide. By editing in this window you can immediately see when you make a line too wide to be correct Fortran.

22.15.6 Fortran Keyword Abbrevs

Fortran mode provides many built-in abbrevs for common keywords and declarations. These are the same sort of abbrev that you can define yourself. To use them, you must turn on Abbrev mode. See Chapter 24 [Abbrevs], page 211.

The built-in abbrevs are unusual in one way: they all start with a semicolon. You cannot normally use semicolon in an abbrev, but Fortran mode makes this possible by changing the syntax of semicolon to "word constituent."

For example, one built-in Fortran abbrev is '`;c`' for '`continue`'. If you insert '`;c`' and then insert a punctuation character such as a space or a newline, the '`;c`' expands automatically to '`continue`', provided Abbrev mode is enabled.

Type '`;?`' or '`;C-h`' to display a list of all the built-in Fortran abbrevs and what they stand for.

22.16 Asm Mode

Asm mode is a major mode for editing files of assembler code. It defines these commands:

`TAB` `tab-to-tab-stop`.

`LFD` Insert a newline and then indent using `tab-to-tab-stop`.

`:` Insert a colon and then remove the indentation from before the label preceding colon. Then do `tab-to-tab-stop`.

`;` Insert or align a comment.

The variable `asm-comment-char` specifies which character starts comments in assembler syntax.

23 Compiling and Testing Programs

The previous chapter discusses the Emacs commands that are useful for making changes in programs. This chapter deals with commands that assist in the larger process of developing and maintaining programs.

23.1 Running 'make', or Compilers Generally

Emacs can run compilers for noninteractive languages such as C and Fortran as inferior processes, feeding the error log into an Emacs buffer. It can also parse the error messages and show you the source lines where compilation errors occurred.

`M-x compile`
: Run a compiler asynchronously under Emacs, with error messages to '`*compilation*`' buffer.

`M-x grep`
: Run `grep` asynchronously under Emacs, with matching lines listed in the buffer named '`*grep*`'.

`M-x kill-compilation`
`M-x kill-grep`
: Kill the running compilation or `grep` subprocess.

`C-x ‘`
: Visit the locus of the next compiler error message or `grep` match.

To run `make` or another compilation command, do `M-x compile`. This command reads a shell command line using the minibuffer, and then executes the command in an inferior shell, putting output in the buffer named '`*compilation*`'. The current buffer's default directory is used as the working directory for the execution of the command; normally, therefore, the compilation happens in this directory.

When the shell command line is read, the minibuffer appears containing a default command line, which is the command you used the last time you did `M-x compile`. If you type just `RET`, the same command line is used again. For the first `M-x compile`, the default is '`make -k`'.

The default compilation command comes from the variable `compile-command`; if the appropriate compilation command for a file is something other than '`make -k`', it can be useful for the file to specify a local value for `compile-command` (see Section 31.2.5 [File Variables], page 297).

Starting a compilation displays the buffer '`*compilation*`' in another window but does not select it. The buffer's mode line tells you whether compilation is finished, with the word '`run`' or '`exit`' inside the parentheses. You do not have to keep this buffer visible; compilation continues in any case. While a compilation is going on, the string '`Compiling`' appears in the mode lines of all windows. When this string disappears, the compilation is finished.

To kill the compilation process, do M-x kill-compilation. When the compiler process terminates, the mode line of the '*compilation*' buffer changes to say 'signal' instead of 'run'. Starting a new compilation also kills any running compilation, as only one can exist at any time. However, M-x compile asks for confirmation before actually killing a compilation that is running.

The '*compilation*' buffer uses a special major mode, Compilation mode. This mode provides the keys SPC and DEL to scroll by screenfuls, and M-n and M-p to move to the next or previous error message. You can also use M-{ and M-} to move up or down to an error message for a different source file. You can visit the source for any particular error message by moving point in '*compilation*' to that error message and typing C-c C-c (compile-goto-error).

To parse the compiler error messages sequentially, type C-x ` (next-error). The character following the C-x is the backquote or "grave accent," not the single-quote. This command is available in all buffers, not just in '*compilation*'. It displays the next error message at the top of one window and source location of the error in another window.

The first time C-x ` is used after the start of a compilation, it moves to the first error's location. Subsequent uses of C-x ` advance down the data set up by the first use. When the preparsed error messages are exhausted, the next C-x ` checks for any more error messages that have come in; this is useful if you start editing the compilation errors while the compilation is still going on. If no more error messages have come in, C-x ` reports an error.

C-u C-x ` discards the preparsed error message data and parses the '*compilation*' buffer over again, then displaying the first error. This way, you can process the same set of errors again.

Instead of running a compiler, you can run grep and see the lines on which matches were found. To do this, type M-x grep with an argument line that contains the same arguments you would give grep when running it normally: a grep-style regexp (usually in single-quotes to quote the shell's special characters) followed by file names which may use wildcards. The output from grep goes in the '*grep*' buffer and the lines that matched can be found with C-x ` as if they were compilation errors.

Note: a shell is used to run the compile command, but the shell is told that it should be noninteractive. This means in particular that the shell starts up with no prompt. If you find your usual shell prompt making an unsightly appearance in the '*compilation*' buffer, it means you have made a mistake in your shell's init file ('.cshrc' or '.shrc' or ...) by setting the prompt unconditionally. The shell init file should set the prompt only if there already is a prompt. In `csh`, here is how to do it:

```
if ($?prompt) set prompt = ...
```

23.2 Running Debuggers Under Emacs

The GUD (Grand Unified Debugger) library provides an interface to various symbolic debuggers from within Emacs. We recommend the debugger GDB, which is free software, but you can also run DBX, SDB or XDB if you have them.

23.2.1 Starting GUD

There are four commands for starting a debugger, each corresponding to a particular debugger program.

`M-x gdb RET` *file* `RET`
: Run GDB as a subprocess of Emacs. This command creates a buffer for input and output to GDB, and switches to it. If a GDB buffer already exists, it just switches to that buffer.

`M-x dbx RET` *file* `RET`
: Similar, but run DBX instead of GDB.

`M-x xdb RET` *file* `RET`
: Similar, but run XDB instead of GDB. Use the variable `gud-xdb-directories` to specify directories to search for source files.

`M-x sdb RET` *file* `RET`
: Similar, but run SDB instead of GDB.

  Some versions of SDB do not mention source file names in their messages. When you use them, you need to have a valid tags table (see Section 22.12 [Tags], page 179) in order for GUD to find functions in the source code. If you have not visited a tags table or the tags table doesn't list one of the functions, you get a message saying '`The sdb support requires a valid tags table to work`'. If this happens, generate a valid tags table in the working directory and try again.

You can only run one debugger process at a time.

Each of these commands takes one argument: a command line to invoke the debugger. In the simplest case, specify just the name of the executable file you want to debug. You may also use options that the debugger supports. However, shell wild cards and variables are not allowed. GUD assumes that the first argument not preceded by a '`-`' is the executable file name.

23.2.2 Debugger Operation

When you run a debugger with GUD, the debugger uses an Emacs buffer for its ordinary input and output. This is called the GUD buffer. The debugger uses other Emacs buffers to display the source files of the program. An arrow (`=>`) in one of these buffers indicates the current execution line. Moving point in this buffer does not move the arrow.

You can start editing these source files at any time in the buffers that were made to display them. The arrow is not part of the file's text; it appears only on the screen. If you do modify a source file, keep in mind that inserting or deleting lines will throw off the arrow's positioning; GUD has no way of figuring out which line corresponded before your changes to the line number in a debugger message. Also, you'll typically have to recompile and restart the program for your changes to be reflected in the debugger's tables.

If you wish, you can control your debugger process entirely through the debugger buffer, which uses a variant of Shell mode. All the usual commands for your debugger are available, and you can use the Shell mode history commands to repeat them.

23.2.3 Commands of GUD

GUD provides a command available in all buffers for setting breakpoints. This command is defined globally because you need to use it in the source files' buffers.

`C-x SPC` Set a breakpoint on the line that point is on.

Here are the other commands provided by GUD. The keys starting with `C-c` are available only in the GUD interaction buffer. The bindings that start with `C-x C-a` are available in the GUD buffer and also in source files.

`C-c C-l`
`C-x C-a C-l`
: Display in another window the last line referred to in the GUD buffer (that is, the line indicated in the last location message). This runs the command `gud-refresh`.

`C-c C-s`
`C-x C-a C-s`
: Execute a single line of code (`gud-step`). If the code contains a function call, execution stops after entering the called function.

`C-c C-n`
`C-x C-a C-n`
: Execute a single line of code, stepping across entire function calls at full speed (`gud-next`).

`C-c C-i`
`C-x C-a C-i`
: Execute a single machine instruction (`gud-stepi`).

`C-c C-c`
`C-x C-a C-c`
: Continue execution until the next breakpoint, or other event that would normally stop the program (`gud-cont`).

`C-c C-d`
`C-x C-a C-d`
: Delete the breakpoint(s) on the current source line, if any (`gud-remove`). If you use this command in the GUD interaction buffer, it applies to the line where the program last stopped.

`C-c C-t`
`C-x C-a C-t`
: Set a temporary breakpoint on the current source line, if any. If you use this command in the GUD interaction buffer, it applies to the line where the program last stopped.

The above commands are common to all supported debuggers. If you are using GDB or (some versions of) DBX, these additional commands are available:

`C-c <`
`C-x C-a <` Select the next enclosing stack frame (`gud-up`). This is equivalent to the '`up`' command.

`C-c >`
`C-x C-a >` Select the next inner stack frame (`gud-down`). This is equivalent to the '`down`' command.

If you are using GDB, two additional keybindings are available:

`C-c C-f`
`C-x C-a C-f`
: Run the program until the selected stack frame returns (or until it stops for some other reason).

These commands interpret a prefix argument as a repeat count, when that makes sense.

23.2.4 GUD Customization

On startup, GUD runs one of the following hooks: `gdb-mode-hook`, if you are using GDB; `dbx-mode-hook`, if you are using DBX; `sdb-mode-hook`, if you are using SDB; and `xdb-mode-hook`, if you are using XDB. You can use these hooks to define custom keybindings for the debugger interaction buffer. See Section 31.2.3 [Hooks], page 294.

Here is a convenient way to define a command that sends a particular command string to the debugger, and set up a key binding for it in the debugger interaction buffer:

```
(gud-def function cmdstring binding docstring)
```

This defines a command named *function* which sends *cmdstring* to the debugger process, with documentation string *docstring*. You can use the command thus defined in any buffer. If *binding* is non-`nil`, `gud-def` also binds the command to `C-c` *binding* in the GUD buffer's mode and to `C-x C-a` *binding* generally.

The command string *cmdstring* may contain certain escape sequences that are filled in with varying data at the time *function* is called:

'`%f`' Name of the current source file. If the current buffer is the GUD buffer, then the "current source file" is the file that the program stopped in.

'`%l`' Number of the current source line. If the current buffer is the GUD buffer, then the "current source line" is the line that the program stopped in.

'`%e`' Text of the C lvalue or function-call expression surrounding point.

'`%a`' Text of the hexadecimal address surrounding point.

'`%p`' Prefix (numeric) argument of the called function (if any) as a number. (If you don't use '`%p`' in the command string, the command you define ignores any prefix argument.)

23.3 Executing Lisp Expressions

Emacs has several different major modes for Lisp and Scheme. They are the same in terms of editing commands, but differ in the commands for executing Lisp expressions.

Emacs-Lisp mode
: The mode for editing source files of programs to run in Emacs Lisp. This mode defines `C-M-x` to evaluate the current defun. See Section 23.4 [Lisp Libraries], page 205.

Lisp Interaction mode
: The mode for an interactive session with Emacs Lisp. It defines `LFD` to evaluate the sexp before point and insert its value in the buffer. See Section 23.6 [Lisp Interaction], page 208.

Lisp mode
: The mode for editing source files of programs that run in Lisps other than Emacs Lisp. This mode defines `C-M-x` to send the current defun to an inferior Lisp process. See Section 23.7 [External Lisp], page 208.

Inferior Lisp mode
: The mode for an interactive session with an inferior Lisp process. This mode combines the special features of Lisp mode and Shell mode (see Section 30.3.3 [Shell Mode], page 278).

Scheme mode
: Like Lisp mode but for Scheme programs.

Inferior Scheme mode
: The mode for an interactive session with an inferior Scheme process.

The editing commands for working with Lisp programs are in fact available globally. See Chapter 22 [Programs], page 163.

23.4 Libraries of Lisp Code for Emacs

Lisp code for Emacs editing commands is stored in files whose names conventionally end in '`.el`'. This ending tells Emacs to edit them in Emacs-Lisp mode (see Section 23.3 [Executing Lisp], page 204).

To execute a file of Emacs Lisp code, use `M-x load-file`. This command reads a file name using the minibuffer and then executes the contents of that file as Lisp code. It is not necessary to visit the file first; in any case, this command reads the file as found on disk, not text in an Emacs buffer.

Once a file of Lisp code is installed in the Emacs Lisp library directories, users can load it using `M-x load-library`. Programs can load it by calling `load-library`, or with `load`, a more primitive function that is similar but accepts some additional arguments.

`M-x load-library` differs from `M-x load-file` in that it searches a sequence of directories and tries three file names in each directory. Suppose your argument is *lib*; the three names are '*lib*`.elc`', '*lib*`.el`', and lastly just '*lib*'. If '*lib*`.elc`' exists, it is by convention the result of compiling '*lib*`.el`'; it is better to load the compiled file, since it will load and run faster.

If `load-library` finds that '*lib*`.el`' is newer than '*lib*`.elc`' file, it prints a warning, because it's likely that somebody made changes to the '`.el`' file and forgot to recompile it.

Because the argument to `load-library` is usually not in itself a valid file name, file name completion is not available. Indeed, when using this command, you usually do not know exactly what file name will be used.

The sequence of directories searched by `M-x load-library` is specified by the variable `load-path`, a list of strings that are directory names. The default value of the list contains the directory where the Lisp code for Emacs itself is stored. If you have libraries of your own, put them in a single directory and add that directory to `load-path`. `nil` in this list stands for the current default directory, but it is probably not a good idea to put `nil` in the list. If you find yourself wishing that `nil` were in the list, most likely what you really want to do is use `M-x load-file` this once.

Often you do not have to give any command to load a library, because the commands defined in the library are set up to *autoload* that library. Running any of those commands causes `load` to be called to load the library; this replaces the autoload definitions with the real ones from the library.

If autoloading a file does not finish, either because of an error or because of a `C-g` quit, all function definitions made by the file are undone automatically. So are any calls to `provide`. As a consequence, if you use one of the autoloadable commands again, the entire file will be loaded a second time. This prevents problems where the command is no longer autoloading but it works wrong because not all the file was loaded. Function definitions are undone only for autoloading; explicit calls to `load` do not undo anything if loading is not completed.

Emacs Lisp code can be compiled into byte-code which loads faster, takes up less space when loaded, and executes faster. See section "Byte Compilation" in *the Emacs Lisp Reference Manual*.

By convention, the compiled code for a library goes in a separate file whose name consists of the library source file with '`c`' appended. Thus, the compiled code for '`foo.el`' goes in '`foo.elc`'. That's why `load-library` searches for '`.elc`' files first.

23.5 Evaluating Emacs-Lisp Expressions

Lisp programs intended to be run in Emacs should be edited in Emacs-Lisp mode; this happens automatically for file names ending in '`.el`'. By contrast, Lisp mode itself is used for editing Lisp programs intended for other Lisp systems. To switch to Emacs-Lisp mode explicitly, use the command `M-x emacs-lisp-mode`.

For testing of Lisp programs to run in Emacs, it is often useful to evaluate part of the program as it is found in the Emacs buffer. For example, after changing the text of a Lisp function definition, evaluating the definition installs the change for future calls to the function. Evaluation of Lisp expressions is also useful in any kind of editing, for invoking noninteractive functions (functions that are not commands).

`M-ESC` Read a single Lisp expression in the minibuffer, evaluate it, and print the value in the echo area (`eval-expression`).

`C-x C-e` Evaluate the Lisp expression before point, and print the value in the echo area (`eval-last-sexp`).

`C-M-x` Evaluate the defun containing or after point, and print the value in the echo area (`eval-defun`).

`M-x eval-region`
Evaluate all the Lisp expressions in the region.

`M-x eval-current-buffer`
Evaluate all the Lisp expressions in the buffer.

`M-ESC` (`eval-expression`) is the most basic command for evaluating a Lisp expression interactively. It reads the expression using the minibuffer, so you can execute any expression on a buffer regardless of what the buffer contains. When the expression is evaluated, the current buffer is once again the buffer that was current when `M-ESC` was typed.

`M-ESC` can easily confuse users who do not understand it, especially on keyboards with autorepeat where it can result from holding down the `ESC` key for too long. Therefore, `eval-expression` is normally a disabled command. Attempting to use this command asks for confirmation and gives you the option of enabling it; once you enable the command, confirmation will no longer be required for it. See Section 31.4.9 [Disabling], page 310.

In Emacs-Lisp mode, the key `C-M-x` is bound to the command `eval-defun`, which parses the defun containing or following point as a Lisp expression and evaluates it. The value is printed in the echo area. This command is convenient for installing in the Lisp environment changes that you have just made in the text of a function definition.

The command `C-x C-e` (`eval-last-sexp`) performs a similar job but is available in all major modes, not just Emacs-Lisp mode. It finds the sexp before point, reads it as a Lisp expression, evaluates it, and prints the value in the echo area. It is sometimes useful to type in an expression and then, with point still after it, type `C-x C-e`.

If `C-M-x` or `C-x C-e` is given a numeric argument, it prints the value by insertion into the current buffer at point, rather than in the echo area. The argument value does not matter.

The most general command for evaluating Lisp expressions from a buffer is `eval-region`. `M-x eval-region` parses the text of the region as one or more Lisp expressions, evaluating them one by one. `M-x eval-current-buffer` is similar but evaluates the entire buffer. This is a reasonable way to install the contents of a file of Lisp code that you are just ready to test. After finding and fixing a bug, use `C-M-x` on each function that you change, to keep the Lisp world in step with the source file.

23.6 Lisp Interaction Buffers

The buffer '*scratch*' which is selected when Emacs starts up is provided for evaluating Lisp expressions interactively inside Emacs.

Thus, the way to use the '*scratch*' buffer is to insert Lisp expressions at the end, ending each one with LFD so that it will be evaluated. This command reads the Lisp expression before point, evaluates it, and inserts the value in printed representation before point. The result is a complete typescript of the expressions you have evaluated and their values.

The '*scratch*' buffer's major mode is Lisp Interaction mode, which is the same as Emacs-Lisp mode except for the binding of LFD.

The rationale for this feature is that Emacs must have a buffer when it starts up, but that buffer is not useful for editing files since a new buffer is made for every file that you visit. The Lisp interpreter typescript is the most useful thing I can think of for the initial buffer to do. Type `M-x lisp-interaction-mode` to put the current buffer in Lisp Interaction mode.

23.7 Running an External Lisp

Emacs has facilities for running programs in other Lisp systems. You can run a Lisp process as an inferior of Emacs, and pass expressions to it to be evaluated. You can also pass changed function definitions directly from the Emacs buffers in which you edit the Lisp programs to the inferior Lisp process.

To run an inferior Lisp process, type `M-x run-lisp`. This runs the program named `lisp`, the same program you would run by typing `lisp` as a shell command, with both input and output going through an Emacs buffer named '*lisp*'. That is to say, any "terminal output" from Lisp will go into the buffer, advancing point, and any "terminal input" for Lisp comes from text in the buffer. (You can change the name of the Lisp executable file by setting the variable `inferior-lisp-program`.)

To give input to Lisp, go to the end of the buffer and type the input, terminated by RET. The '*lisp*' buffer is in Inferior Lisp mode, which combines the special characteristics of Lisp mode and Shell mode (see Section 30.3.3 [Shell Mode], page 278).

For the source files of programs to run in external Lisps, use Lisp mode. This mode can be selected with `M-x lisp-mode`, and is used automatically for files whose names end in '.l', '.lsp', or '.lisp', as most Lisp systems usually expect.

When you edit a function in a Lisp program you are running, the easiest way to send the changed definition to the inferior Lisp process is the key `C-M-x`. In Lisp mode, this runs the function `lisp-eval-defun`, which finds the defun around or following point and sends it as input to the Lisp process. (Emacs can send input to any inferior process regardless of what buffer is current.)

Contrast the meanings of `C-M-x` in Lisp mode (for editing programs to be run in another Lisp system) and Emacs-Lisp mode (for editing Lisp programs to be run in Emacs): in both modes it has the effect of installing the function definition that point is in, but the way of doing so is different according to where the relevant Lisp environment is found. See Section 23.3 [Executing Lisp], page 204.

24 Abbrevs

An *abbrev* is a word which *expands*, if you insert it, into some different text. Abbrevs are defined by the user to expand in specific ways. For example, you might define '`foo`' as an abbrev expanding to '`find outer otter`'. With this abbrev defined, you would be able to get '`find outer otter` ' into the buffer by typing `f o o SPC`.

Abbrevs expand only when Abbrev mode (a minor mode) is enabled. Disabling Abbrev mode does not cause abbrev definitions to be forgotten, but they do not expand until Abbrev mode is enabled again. The command `M-x abbrev-mode` toggles Abbrev mode; with a numeric argument, it turns Abbrev mode on if the argument is positive, off otherwise. See Section 31.1 [Minor Modes], page 291. `abbrev-mode` is also a variable; Abbrev mode is on when the variable is non-`nil`. The variable `abbrev-mode` automatically becomes local to the current buffer when it is set.

Abbrev definitions can be *mode-specific*—active only in one major mode. Abbrevs can also have *global* definitions that are active in all major modes. The same abbrev can have a global definition and various mode-specific definitions for different major modes. A mode specific definition for the current major mode overrides a global definition.

Abbrevs can be defined interactively during the editing session. Lists of abbrev definitions can also be saved in files and reloaded in later sessions. Some users keep extensive lists of abbrevs that they load in every session.

A second kind of abbreviation facility is called the *dynamic expansion*. Dynamic abbrev expansion happens only when you give an explicit command and the result of the expansion depends only on the current contents of the buffer. See Section 24.5 [Dynamic Abbrevs], page 215.

24.1 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`
After this command, there are no abbrev definitions in effect.

The usual way to define an abbrev is to enter the text you want the abbrev to expand to, position point after it, and type `C-x 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 of the abbrev you want is already in the buffer instead of the 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 argument found in the buffer and the argument read using the minibuffer.

To change the definition of an abbrev, just add the new definition. You will be asked to confirm if the abbrev has a prior definition. To remove an abbrev definition, give a negative argument to `C-x a g` or `C-x a l`. You must choose the command to specify whether to kill a global definition or a mode-specific definition for the current mode, since those two definitions are independent for one abbrev.

`M-x kill-all-abbrevs` removes all the abbrev definitions there are.

24.2 Controlling Abbrev Expansion

An abbrev expands whenever it is present in the buffer just before point and a self-inserting punctuation character (`SPC`, comma, etc.) is typed. Most often the way an abbrev is used is to insert the abbrev followed by punctuation.

Abbrev expansion preserves case; thus, '`foo`' expands into '`find outer otter`'; '`Foo`' into '`Find outer otter`', and '`FOO`' into '`FIND OUTER OTTER`' or '`Find Outer Otter`' according to the variable `abbrev-all-caps` (a non-`nil` value chooses the first of the two expansions).

These two commands are used to control abbrev expansion:

`M-'`
: Separate a prefix from a following abbrev to be expanded (`abbrev-prefix-mark`).

`C-x a e`
: Expand the abbrev before point (`expand-abbrev`). This is effective even when Abbrev mode is not enabled.

`M-x unexpand-abbrev`
Undo last abbrev expansion.

`M-x expand-region-abbrevs`
Expand some or all abbrevs found in the region.

You may wish to expand an abbrev with a prefix attached; for example, if '`cnst`' expands into '`construction`', you might want to use it to enter '`reconstruction`'. It does not work to type `recnst`, because that is not necessarily a defined abbrev. What does work is to use the command `M-'` (`abbrev-prefix-mark`) in between the prefix '`re`' and the abbrev '`cnst`'. First, insert '`re`'. Then type `M-'`; this inserts a minus sign in the buffer to indicate that it has done its work. Then insert the abbrev '`cnst`'; the buffer now contains '`re-cnst`'. Now insert a punctuation character to expand the abbrev '`cnst`' into '`construction`'. The minus sign is deleted at this point, because `M-'` left word for this to be done. The resulting text is the desired '`reconstruction`'.

If you actually want the text of the abbrev in the buffer, rather than its expansion, you can accomplish this by inserting the following punctuation with `C-q`. Thus, `foo C-q -` leaves '`foo-`' in the buffer.

If you expand an abbrev by mistake, you can undo the expansion (replace the expansion by the original abbrev text) with `M-x unexpand-abbrev`. `C-_` (`undo`) can also be used to undo the expansion; but first it undoes the insertion of the following punctuation character. Therefore, if you want both the punctuation character and the unexpanded abbrev, you must reinsert the punctuation character, quoting it with `C-q`.

`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 31.2.3 [Hooks], page 294).

24.3 Examining and Editing Abbrevs

`M-x list-abbrevs`
Print a list of all abbrev definitions.

`M-x edit-abbrevs`
Edit a list of abbrevs; you can add, alter or remove definitions.

The output from `M-x list-abbrevs` looks like this:

```
(lisp-mode-abbrev-table)
"dk"        0    "define-key"
(global-abbrev-table)
"dfn"        0    "definition"
```

(Some blank lines of no semantic significance, and some other abbrev tables, have been omitted.)

A line containing a name in parentheses is the header for abbrevs in a particular abbrev table; `global-abbrev-table` contains all the global abbrevs, and the other abbrev tables that are named after major modes contain the mode-specific abbrevs.

Within each abbrev table, each nonblank line defines one abbrev. The word at the beginning is the abbrev. The number that appears is the number of times the abbrev has been expanded. Emacs keeps track of this to help you see which abbrevs you actually use, in case you decide to eliminate those that you don't use often. The string at the end of the line is the expansion.

`M-x edit-abbrevs` allows you to add, change or kill abbrev definitions by editing a list of them in an Emacs buffer. The list has the same format described above. The buffer of abbrevs is called '`*Abbrevs*`', and is in Edit-Abbrevs mode. This mode redefines the key `C-c C-c` to install the abbrev definitions as specified in the buffer. The command that does this is `edit-abbrevs-redefine`. Any abbrevs not described in the buffer are eliminated when this is done.

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.

24.4 Saving Abbrevs

These commands allow you to keep abbrev definitions between editing sessions.

`M-x write-abbrev-file`
: Write a file describing all defined abbrevs.

`M-x read-abbrev-file`
: Read such a file and define abbrevs as specified there.

`M-x quietly-read-abbrev-file`
: Similar but do not display a message about what is going on.

`M-x define-abbrevs`
: Define abbrevs from 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 writes a description of all current abbrev definitions into that file. The text stored in the file looks like the output of `M-x list-abbrevs`. This is used to save abbrev definitions for use in a later session.

`M-x read-abbrev-file` reads a file name using the minibuffer and reads the file, defining abbrevs according to the contents of the file. `M-x quietly-read-abbrev-file` is the same except that it does not display a message in the echo area saying that it is doing its work; it is actually useful primarily in the '`.emacs`' file. If an empty argument is given to either of these functions, the file name used is the value of the variable `abbrev-file-name`, which is by default `"~/.abbrev_defs"`.

Emacs will offer to save abbrevs automatically if you have changed any of them, whenever it offers to save all files (for `C-x s` or `C-x C-c`). This feature can be inhibited by setting the variable `save-abbrevs` to `nil`.

The commands `M-x insert-abbrevs` and `M-x define-abbrevs` are similar to the previous commands but work on text in an Emacs buffer. `M-x insert-abbrevs` inserts text into the current buffer before point, describing all current abbrev definitions; `M-x define-abbrevs` parses the entire current buffer and defines abbrevs accordingly.

24.5 Dynamic Abbrev Expansion

The abbrev facility described above operates automatically as you insert text, but all abbrevs must be defined explicitly. By contrast, *dynamic abbrevs* allow the meanings of abbrevs to be determined automatically from the contents of the buffer, but dynamic abbrev expansion happens only when you request it explicitly.

`M-/`
: Expand the word in the buffer before point as a *dynamic abbrev*, by searching in the buffer for words starting with that abbreviation (`dabbrev-expand`).

For example, if the buffer contains '`does this follow `' and you type `f o M-/`, the effect is to insert '`follow`' because that is the last word in the buffer that starts with '`fo`'. A numeric argument to `M-/` says to take the second, third, etc. distinct expansion found looking backward from point. Repeating `M-/` searches for an alternative expansion by looking farther back. After the entire buffer before point has been considered, the buffer after point is searched.

Dynamic abbrev expansion is completely independent of Abbrev mode; the expansion of a word with `M-/` is completely independent of whether it has a definition as an ordinary abbrev.

25 Editing Pictures

If you want to create a picture made out of text characters (for example, a picture of the division of a register into fields, as a comment in a program), use the command `M-x edit-picture` to enter Picture mode.

In Picture mode, editing is based on the *quarter-plane* model of text, according to which the text characters lie studded on an area that stretches infinitely far to the right and downward. The concept of the end of a line does not exist in this model; the most you can say is where the last nonblank character on the line is found.

Of course, Emacs really always considers text as a sequence of characters, and lines really do have ends. But in Picture mode most frequently-used keys are rebound to commands that simulate the quarter-plane model of text. They do this by inserting spaces or by converting tabs to spaces.

Most of the basic editing commands of Emacs are redefined by Picture mode to do essentially the same thing but in a quarter-plane way. In addition, Picture mode defines various keys starting with the `C-c` prefix to run special picture editing commands.

One of these keys, `C-c C-c`, is pretty important. Often a picture is part of a larger file that is usually edited in some other major mode. `M-x edit-picture` records the name of the previous major mode so you can use the `C-c C-c` command (`picture-mode-exit`) later to go back to that mode. `C-c C-c` also deletes spaces from the ends of lines, unless given a numeric argument.

The commands used in Picture mode all work in other modes (provided the '`picture`' library is loaded), but are not bound to keys except in Picture mode. Note that the descriptions below talk of moving "one column" and so on, but all the picture mode commands handle numeric arguments as their normal equivalents do.

Turning on Picture mode runs the hook `picture-mode-hook` (see Section 31.2.3 [Hooks], page 294).

25.1 Basic Editing in Picture Mode

Most keys do the same thing in Picture mode that they usually do, but do it in a quarter-plane style. For example, `C-f` is rebound to run `picture-forward-column`, a command which moves point one column to the right, inserting a space if necessary so that the actual end of the line makes no difference. `C-b` is rebound to run `picture-backward-column`, which always moves point left one column, converting a tab to multiple spaces if necessary. `C-n` and `C-p` are rebound to run `picture-move-down` and `picture-move-up`, which can either insert spaces or convert tabs as necessary to make sure that point stays in exactly the same column. `C-e` runs `picture-end-of-line`, which moves to after the last nonblank character on the line. There is no need to change `C-a`, as the choice of screen model does not affect beginnings of lines.

Insertion of text is adapted to the quarter-plane screen model through the use of Overwrite mode (see Section 31.1 [Minor Modes], page 291). Self-inserting characters replace existing text, column by column, rather than pushing existing text to the right. `RET` runs `picture-newline`, which just moves to the beginning of the following line so that new text will replace that line.

Picture mode provides erasure instead of deletion and killing of text. `DEL` (`picture-backward-clear-column`) replaces the preceding character with a space rather than removing it; this moves point backwards. `C-d` (`picture-clear-column`) replaces the next character or characters with spaces, but does not move point. (If you want to clear characters to spaces and move forward over them, use `SPC`.) `C-k` (`picture-clear-line`) really kills the contents of lines, but does not delete the newlines from the buffer.

To do actual insertion, you must use special commands. `C-o` (`picture-open-line`) still creates a blank line, but does so after the current line; it never splits a line. `C-M-o`, `split-line`, makes sense in Picture mode, so it is not changed. `LFD` (`picture-duplicate-line`) inserts below the current line another line with the same contents.

If you want to do real deletion in Picture mode, use `C-w`, `C-c C-d` (which is defined as `delete-char`, as `C-d` is in other modes), or one of the picture rectangle commands (see Section 25.4 [Rectangles in Picture], page 219).

25.2 Controlling Motion after Insert

Since "self-inserting" characters in Picture mode just overwrite and move point, there is no essential restriction on how point should be moved. Normally point moves right, but you can specify any of the eight orthogonal or diagonal directions for motion after a "self-inserting" character. This is useful for drawing lines in the buffer.

`C-c <`
: Move left after insertion (`picture-movement-left`).

`C-c >`
: Move right after insertion (`picture-movement-right`).

`C-c ^`
: Move up after insertion (`picture-movement-up`).

`C-c .`
: Move down after insertion (`picture-movement-down`).

`` C-c ` ``
: Move up and left ("northwest") after insertion (`picture-movement-nw`).

`C-c '`
: Move up and right ("northeast") after insertion (`picture-movement-ne`).

`C-c /` Move down and left ("southwest") after insertion (`picture-movement-sw`).

`C-c \` Move down and right ("southeast") after insertion (`picture-movement-se`).

Two motion commands move based on the current Picture insertion direction. The command `C-c C-f` (`picture-motion`) moves in the same direction as motion after "insertion" currently does, while `C-c C-b` (`picture-motion-reverse`) moves in the opposite direction.

25.3 Picture Mode Tabs

Two kinds of tab-like action are provided in Picture mode. Use `M-TAB` (`picture-tab-search`) for context-based tabbing. With no argument, it moves to a point underneath the next "interesting" character that follows whitespace in the previous nonblank line. "Next" here means "appearing at a horizontal position greater than the one point starts out at". With an argument, as in `C-u M-TAB`, this command moves to the next such interesting character in the current line. `M-TAB` does not change the text; it only moves point. "Interesting" characters are defined by the variable `picture-tab-chars`, which contains a string whose characters are all considered interesting. Its default value is `"!-~"`.

`TAB` itself runs `picture-tab`, which operates based on the current tab stop settings; it is the Picture mode equivalent of `tab-to-tab-stop`. Normally it just moves point, but with a numeric argument it clears the text that it moves over.

The context-based and tab-stop-based forms of tabbing are brought together by the command `C-c TAB`, `picture-set-tab-stops`. This command sets the tab stops to the positions which `M-TAB` would consider significant in the current line. The use of this command, together with `TAB`, can get the effect of context-based tabbing. But `M-TAB` is more convenient in the cases where it is sufficient.

25.4 Picture Mode Rectangle Commands

Picture mode defines commands for working on rectangular pieces of the text in ways that fit with the quarter-plane model. The standard rectangle commands may also be useful (see Section 10.4 [Rectangles], page 63).

`C-c C-k` Clear out the region-rectangle (`picture-clear-rectangle`). With argument, kill it.

`C-c C-w` *r* Similar but save rectangle contents in register *r* first (`picture-clear-rectangle-to-register`).

`C-c C-y` Copy last killed rectangle into the buffer by overwriting, with upper left corner at point (`picture-yank-rectangle`). With argument, insert instead.

`C-c C-x` *r* Similar, but use the rectangle in register *r* (`picture-yank-rectangle-from-register`).

The picture rectangle commands `C-c C-k` (`picture-clear-rectangle`) and `C-c C-w` (`picture-clear-rectangle-to-register`) differ from the standard rectangle commands in that they normally clear the rectangle instead of deleting it; this is analogous with the way `C-d` is changed in Picture mode.

However, deletion of rectangles can be useful in Picture mode, so these commands delete the rectangle if given a numeric argument.

The Picture mode commands for yanking rectangles differ from the standard ones in overwriting instead of inserting. This is the same way that Picture mode insertion of other text differs from other modes. `C-c C-y` (`picture-yank-rectangle`) inserts (by overwriting) the rectangle that was most recently killed, while `C-c C-x` (`picture-yank-rectangle-from-register`) does likewise for the rectangle found in a specified register.

26 Sending Mail

To send a message in Emacs, you start by typing a command (`C-x m`) to select and initialize the '`*mail*`' buffer. Then you edit the text and headers of the message in this buffer, and type another command (`C-c C-c`) to send the message.

`C-x m` Begin composing a message to send (`mail`).

`C-x 4 m` Likewise, but display the message in another window (`mail-other-window`).

`C-x 5 m` Likewise, but make a new frame (`mail-other-frame`).

`C-c C-c` In Mail mode, send the message and switch to another buffer (`mail-send-and-exit`).

The command `C-x m` (`mail`) selects a buffer named '`*mail*`' and initializes it with the skeleton of an outgoing message. `C-x 4 m` (`mail-other-window`) selects the '`*mail*`' buffer in a different window, leaving the previous current buffer visible. `C-x 5 m` (`mail-other-frame`) creates a new frame to select the '`*mail*`' buffer.

Because the mail composition buffer is an ordinary Emacs buffer, you can switch to other buffers while in the middle of composing mail, and switch back later (or never). If you use the `C-x m` command again when you have been composing another message but have not sent it, you are asked to confirm before the old message is erased. If you answer `n`, the '`*mail*`' buffer is left selected with its old contents, so you can finish the old message and send it. `C-u C-x m` is another way to do this. Sending the message marks the '`*mail*`' buffer "unmodified", which avoids the need for confirmation when `C-x m` is next used.

If you are composing a message in the '`*mail*`' buffer and want to send another message before finishing the first, rename the '`*mail*`' buffer using `M-x rename-uniquely` (see Section 16.3 [Misc Buffer], page 120). Then you can use `C-x m` or its variants described above to make a new '`*mail`' buffer. Once you've done that, you can work with each mail buffer independently.

26.1 The Format of the Mail Buffer

In addition to the *text* or *body*, a message has *header fields* which say who sent it, when, to whom, why, and so on. Some header fields such as the date and sender are created automatically after the message is sent. Others, such as the recipient names, must be specified by you in order to send the message properly.

Mail mode provides a few commands to help you edit some header fields, and some are preinitialized in the buffer automatically at times. You can insert or edit any header fields using ordinary editing commands.

The line in the buffer that says

```
--text follows this line--
```

is a special delimiter that separates the headers you have specified from the text. Whatever follows this line is the text of the message; the headers precede it. The delimiter line itself does not appear in the message actually sent. The text used for the delimiter line is controlled by the variable `mail-header-separator`.

Here is an example of what the headers and text in the '`*mail*`' buffer might look like.

```
To: gnu@prep.ai.mit.edu
CC: lungfish@spam.org, byob@spam.org
Subject: The Emacs Manual
--Text follows this line--
Please ignore this message.
```

26.2 Mail Header Fields

There are several header fields you can use in the '`*mail*`' buffer. Each header field starts with a field name at the beginning of a line, terminated by a colon. Upper and lower case are equivalent in field names (and in mailing addresses also). After the colon and optional whitespace comes the contents of the field.

'`To`' This field contains the mailing addresses to which the message is addressed.

'`Subject`' The contents of the '`Subject`' field should be a piece of text that says what the message is about. The reason '`Subject`' fields are useful is that most mail-reading programs can provide a summary of messages, listing the subject of each message but not its text.

'`CC`' This field contains additional mailing addresses to send the message to, but whose readers should not regard the message as addressed to them.

'`BCC`' This field contains additional mailing addresses to send the message to, which should not appear in the header of the message actually sent. Copies sent this way are called *blind carbon copies*.

To send a blind carbon copy of every outgoing message to yourself, set the variable `mail-self-blind` to `t`.

'FCC' This field contains the name of one file (in system mail file format) to which a copy of the message should be appended when the message is sent. Do not output directly into an Rmail file with FCC; instead, output to an inbox file and "get new mail" from that inbox file into the Rmail file. See Section 27.4 [Rmail Inbox], page 230.

'From' Use the 'From' field to say who you are, when the account you are using to send the mail is not your own. The contents of the 'From' field should be a valid mailing address, since replies will normally go there.

'Reply-to'
: Use this field to direct replies to a different address. Most mail-reading programs (including Rmail) automatically send replies to the 'Reply-to' address in preference to the 'From' address. By adding a 'Reply-to' field to your header, you can work around any problems your 'From' address may cause for replies.

  If you want to put the same 'Reply-to' address into every outgoing message, set the variable `mail-default-reply-to` to that address (as a string).

'In-reply-to'
: This field contains a piece of text describing a message you are replying to. Some mail systems can use this information to correlate related pieces of mail. Normally this field is filled in by Rmail when you are replying to a message in Rmail, and you never need to think about it (see Chapter 27 [Rmail], page 227).

The 'To', 'CC', 'BCC' and 'FCC' fields can appear any number of times, to specify many places to send the message.

The 'To', 'CC', and 'BCC' fields can have continuation lines. All the lines starting with whitespace, following the line on which the field starts, are considered part of the field. For example,

```
To: foo@here.net, this@there.net,
  me@gnu.cambridge.mass.usa.earth.spiral3281
```

If the variable `mail-archive-file-name` is non-`nil`, it should be a string naming a file; every time you start to edit a message to send, the message starts out with an 'FCC' field specifying that file. Unless you remove the 'FCC' field before sending, the message will be written into that file when it is sent.

26.3 Mail Aliases

You can define *mail aliases* in a file named '~/.mailrc'. These are short mnemonic names stand for mail addresses or groups of mail addresses. Like many other mail programs, Emacs expands aliases when they occur in the 'To', 'CC', and 'BCC' fields.

To define an alias in '~/.mailrc', write a line in the following format:

```
alias shortaddress fulladdress
```

For instance, if you send want to use maingnu as a short form of the address gnu@prep.ai.mit.edu, put in the line:

```
alias maingnu gnu@prep.ai.mit.edu
```

The file '~/.mailrc' is used primarily by other mail-reading programs; it can contain various other commands. Emacs ignores everything in it except for alias definitions.

Another way to define a mail alias, within Emacs alone, is with the define-mail-alias command. It prompts for the alias and then the full address. You can use it to define aliases in your '.emacs' file, like this:

```
(define-mail-alias "maingnu" "gnu@prep.ai.mit.edu")
```

define-mail-alias records aliases by adding them to a variable named mail-aliases. If your are comfortable with manipulating Lisp lists, you can set mail-aliases directly. The initial value of mail-aliases is t, which means that Emacs should read '.mailrc' to get the proper value.

Normally, Emacs expands aliases when you send the message. If you like, you can have mail aliases expand as abbrevs, as soon as you type them in. To enable this feature, execute the following:

```
(add-hook 'mail-setup-hook 'mail-abbrevs-setup)
```

This can go in your '.emacs' file. See Section 31.2.3 [Hooks], page 294. If you use this feature, you must use define-mail-abbrev instead of define-mail-alias; the latter does not work with this package. Also, the mail abbreviation package uses the variable mail-abbrevs instead of mail-aliases.

Note that abbrevs expand only if you insert a word-separator character afterward. However, any mail aliases that you didn't expand in the mail buffer are expanded subsequently when you send the message. See Chapter 24 [Abbrevs], page 211.

26.4 Mail Mode

The major mode used in the '*mail*' buffer is Mail mode, which is much like Text mode except that various special commands are provided on the C-c prefix. These commands all have to do specifically with editing or sending the message.

C-c C-s
: Send the message, and leave the '*mail*' buffer selected (mail-send).

`C-c C-c` Send the message, and select some other buffer (`mail-send-and-exit`).

`C-c C-f C-t`
: Move to the '`To`' header field, creating one if there is none (`mail-to`).

`C-c C-f C-s`
: Move to the '`Subject`' header field, creating one if there is none (`mail-subject`).

`C-c C-f C-c`
: Move to the '`CC`' header field, creating one if there is none (`mail-cc`).

`C-c C-w` Insert the file '`~/.signature`' at the end of the message text (`mail-signature`).

`C-c C-y` Yank the selected message from Rmail (`mail-yank-original`). This command does nothing unless your command to start sending a message was issued with Rmail.

`C-c C-q` Fill all paragraphs of yanked old messages, each individually (`mail-fill-yanked-message`).

There are two ways to send the message. `C-c C-s` (`mail-send`) sends the message and marks the '`*mail*`' buffer unmodified, but leaves that buffer selected so that you can modify the message (perhaps with new recipients) and send it again. `C-c C-c` (`mail-send-and-exit`) sends and then deletes the window or switches to another buffer. It puts the '`*mail*`' buffer at the lowest priority for reselection by default, since you are finished with using it. This is the usual way to send the message.

Mail mode provides some other special commands that are useful for editing the headers and text of the message before you send it. There are three commands defined to move point to particular header fields, all based on the prefix `C-c C-f` ('`C-f`' is for "field"). They are `C-c C-f C-t` (`mail-to`) to move to the '`To`' field, `C-c C-f C-s` (`mail-subject`) for the '`Subject`' field, and `C-c C-f C-c` (`mail-cc`) for the '`CC`' field. These fields have special motion commands because they are the most common fields for the user to want to edit.

`C-c C-w` (`mail-signature`) adds a standard piece text at the end of the message to say more about who you are. The text comes from the file '`.signature`' in your home directory.

To insert signatures automatically, set the variable `mail-signature` non-`nil`; then starting a mail message automatically inserts the contents of your '`.signature`' file. If you want to omit your signature from a particular message, delete it from the buffer before you send the message.

When mail sending is invoked from the Rmail mail reader using an Rmail command, `C-c C-y` can be used inside the '`*mail*`' buffer to insert the text of the message you are replying to. Normally it indents each line of that message four spaces and eliminates most header fields. A numeric argument specifies the number of spaces to indent. An argument of just `C-u` says not to indent at all and not to eliminate anything. `C-c C-y` always uses the current message from the '`RMAIL`' buffer, so you can insert several old messages by selecting one in '`RMAIL`', switching to '`*mail*`' and yanking it, then switching back to '`RMAIL`' to select another.

You can specify the text for `C-c C-y` to insert at the beginning of each line: set `mail-yank-prefix` to the desired string. (A value of `nil` means to use indentation; this is the default.) However, `C-u C-c C-y` never adds anything at the beginning of the inserted lines, regardless of the value of `mail-yank-prefix`.

After using `C-c C-y`, you can use the command `C-c C-q` (`mail-fill-yanked-message`) to fill the paragraphs of the yanked old message or messages. One use of `C-c C-q` fills all such paragraphs, each one individually.

Mail mode defines the character '`%`' as part of a word; this is helpful for using the word commands to edit mail addresses.

Turning on Mail mode (which `C-x m` does automatically) runs the normal hooks `text-mode-hook` and `mail-mode-hook`. Initializing a new outgoing message runs the normal hook `mail-setup-hook`; if you want to add special fields to your mail header or make other changes to the appearance of the mail buffer, use that hook. See Section 31.2.3 [Hooks], page 294.

The main difference between these hooks is just when they are invoked. Whenever you type `M-x mail`, `mail-mode-hook` runs as soon as the '`*mail*`' buffer is created. Then the `mail-setup` function puts in the default contents of the buffer. After these default contents are inserted, `mail-setup-hook` runs.

26.5 Distracting the NSA

`M-x spook` adds a line of randomly chosen keywords to an outgoing mail message. The keywords are chosen from a list of words that suggest you are discussing something subversive.

The idea behind this feature is that the suspicion that the NSA snoops on all electronic mail messages that contain keywords suggesting they might be interested. (The NSA says they don't, but there is no way to be certain.) The idea is that if lots of people add suspicious words to their messages, the NSA will get so busy with spurious input that they will have to give up reading it all.

Whether or not this confuses the NSA, it at least amuses people.

27 Reading Mail with Rmail

Rmail is an Emacs subsystem for reading and disposing of mail that you receive. Rmail stores mail messages in files called Rmail files. Reading the message in an Rmail file is done in a special major mode, Rmail mode, which redefines most letters to run commands for managing mail. To enter Rmail, type `M-x rmail`. This reads your primary mail file, merges new mail in from your inboxes, displays the first message you haven't read yet, and lets you begin reading.

Using Rmail in the simplest fashion, you have one Rmail file '`~/RMAIL`' in which all of your mail is saved. It is called your *primary mail file*. In more sophisticated usage, you can copy messages into other Rmail files and then edit those files with Rmail.

Rmail uses narrowing to hide all but one message in the Rmail file. The message that is shown is called the *current message*. Rmail mode's special commands can do such things as delete the current message, copy into another file, send a reply, or move to another message.

Within the Rmail file, messages are arranged sequentially in order of receipt. They are also assigned consecutive integers as their *message numbers*. The number of the current message is displayed in Rmail's mode line, followed by the total number of messages in the file. You can move to a message by specifying its message number using the `j` key (see Section 27.2 [Rmail Motion], page 228).

Following the usual conventions of Emacs, changes in an Rmail file become permanent only when the file is saved. You can do this with `s` (`rmail-save`), which also expunges deleted messages from the file first (see Section 27.3 [Rmail Deletion], page 229). To save the file without expunging, use `C-x C-s`. Rmail saves the Rmail file spontaneously when moving new mail from an inbox file (see Section 27.4 [Rmail Inbox], page 230).

You can exit Rmail with `q` (`rmail-quit`); this expunges and saves the Rmail file and then switches to another buffer. But there is no need to 'exit' formally. If you switch from Rmail to editing in other buffers, and never happen to switch back, you have exited. (The Rmail command `b`, `rmail-bury`, does this for you.) Just make sure to save the Rmail file eventually (like any other file you have changed). `C-x s` is a good enough way to do this (see Section 15.3 [Saving], page 96).

27.1 Scrolling Within a Message

When Rmail displays a message that does not fit on the screen, it is necessary to scroll through it. This could be done with `C-v`, `M-v` and `M-<`, but in Rmail scrolling is so frequent that it deserves to be easier to type.

`SPC` Scroll forward (`scroll-up`).

`DEL` Scroll backward (`scroll-down`).

`.` Scroll to start of message (`rmail-beginning-of-message`).

Since the most common thing to do while reading a message is to scroll through it by screenfuls, Rmail makes `SPC` and `DEL` synonyms of `C-v` (`scroll-up`) and `M-v` (`scroll-down`)

The command `.` (`rmail-beginning-of-message`) scrolls back to the beginning of the selected message. This is not quite the same as `M-<`: for one thing, it does not set the mark; for another, it resets the buffer boundaries to the current message if you have changed them.

27.2 Moving Among Messages

The most basic thing to do with a message is to read it. The way to do this in Rmail is to make the message current. The usual practice is to move sequentially through the file, since this is the order of receipt of messages. When you enter Rmail, you are positioned at the first message that you have not yet made current (that is, the first one that has the '`unseen`' attribute; see Section 27.7 [Rmail Labels], page 233). Move forward to see the other new messages; move backward to reexamine old messages.

`n` Move to the next nondeleted message, skipping any intervening deleted messages (`rmail-next-undeleted-message`).

`p` Move to the previous nondeleted message (`rmail-previous-undeleted-message`).

`M-n` Move to the next message, including deleted messages (`rmail-next-message`).

`M-p` Move to the previous message, including deleted messages (`rmail-previous-message`).

`j` Move to the first message. With argument *n*, move to message number *n* (`rmail-show-message`).

`>` Move to the last message (`rmail-last-message`).

`<` Move to the first message (`rmail-first-message`).

`M-s` *regexp* `RET`
Move to the next message containing a match for *regexp* (`rmail-search`).

`- M-s` *regexp* `RET`
Move to the previous message containing a match for *regexp*.

n and p are the usual way of moving among messages in Rmail. They move through the messages sequentially, but skip over deleted messages, which is usually what you want to do. Their command definitions are named rmail-next-undeleted-message and rmail-previous-undeleted-message. If you do not want to skip deleted messages—for example, if you want to move to a message to undelete it—use the variants M-n and M-p (rmail-next-message and rmail-previous-message). A numeric argument to any of these commands serves as a repeat count.

In Rmail, you can specify a numeric argument by typing just the digits. You don't need to type C-u first.

The M-s (rmail-search) command is Rmail's version of search. The usual incremental search command C-s works in Rmail, but it searches only within the current message. The purpose of M-s is to search for another message. It reads a regular expression (see Section 13.5 [Regexps], page 79) nonincrementally, then searches starting at the beginning of the following message for a match. The message containing the match is selected.

If *regexp* is empty, M-s reuses the regexp used the previous time.

To search backward in the file for another message, give M-s a negative argument. In Rmail you can do this with - M-s.

It is also possible to search for a message based on labels. See Section 27.7 [Rmail Labels], page 233.

To move to a message specified by absolute message number, use j (rmail-show-message) with the message number as argument. With no argument, j selects the first message. > (rmail-last-message) selects the last message. < (rmail-first-message) selects the first message.

27.3 Deleting Messages

When you no longer need to keep a message, you can *delete* it. This flags it as ignorable, and some Rmail commands pretend it is no longer present; but it still has its place in the Rmail file, and still has its message number.

Expunging the Rmail file actually removes the deleted messages. The remaining messages are renumbered consecutively. Expunging is the only action that changes the message number of any message, except for undigestifying (see Section 27.11 [Rmail Digest], page 240).

d
: Delete the current message, and move to the next nondeleted message (rmail-delete-forward).

C-d
: Delete the current message, and move to the previous nondeleted message (rmail-delete-backward).

u
: Undelete the current message, or move back to a deleted message and undelete it (rmail-undelete-previous-message).

`x` Expunge the Rmail file (`rmail-expunge`).

There are two Rmail commands for deleting messages. Both delete the current message and select another message. `d` (`rmail-delete-forward`) moves to the following message, skipping messages already deleted, while `C-d` (`rmail-delete-backward`) moves to the previous nondeleted message. If there is no nondeleted message to move to in the specified direction, the message that was just deleted remains current.

To make all the deleted messages finally vanish from the Rmail file, type `x` (`rmail-expunge`). Until you do this, you can still *undelete* the deleted messages.

To undelete, type `u` (`rmail-undelete-previous-message`), which is designed to cancel the effect of a `d` command (usually). It undeletes the current message if the current message is deleted. Otherwise it moves backward to previous messages until a deleted message is found, and undeletes that message.

You can usually undo a `d` with a `u` because the `u` moves back to and undeletes the message that the `d` deleted. But this does not work when the `d` skips a few already-deleted messages that follow the message being deleted; then the `u` command undeletes the last of the messages that were skipped. There is no clean way to avoid this problem. However, by repeating the `u` command, you can eventually get back to the message that you intend to undelete. You can also select a particular deleted message with `M-p` command, then type `u` to undelete it.

A deleted message has the '`deleted`' attribute, and as a result '`deleted`' appears in the mode line when the current message is deleted. In fact, deleting or undeleting a message is nothing more than adding or removing this attribute. See Section 27.7 [Rmail Labels], page 233.

27.4 Rmail Files and Inboxes

Unix places incoming mail for you in a file that we call your *inbox*. When you start up Rmail, it copies the new messages from your inbox into your primary mail file, an Rmail file, which also contains other messages saved from previous Rmail sessions. It is in this file that you actually read the mail with Rmail. This operation is called *getting new mail*. It can be repeated at any time using the `g` key in Rmail. The inbox file name is '`/usr/spool/mail/`*username*' on some systems (typically Berkeley-derived ones) and '`/usr/mail/`*username*' on other systems.

There are three reason for having separate Rmail files and inboxes.

1. The inbox file format varies between operating systems and according to the other mail software in use. Only one part of Rmail needs to know

about the alternatives, and it need only understand how to convert all of them to Rmail's own format.

2. The inbox file format usually doesn't provide a place for all the information that Rmail records.
3. It is very cumbersome to access an inbox file without danger of losing mail, because it is necessary to interlock with mail delivery. Moreover, different operating systems use different interlocking techniques. The strategy of moving mail out of the inbox once and for all into a separate Rmail file avoids the need for interlocking in all the rest of Rmail, since only Rmail operates on the Rmail file.

When getting new mail, Rmail first copies the new mail from the inbox file to the Rmail file; then it saves the Rmail file; then it truncates the inbox file. This way, a system crash may cause duplication of mail between the inbox and the Rmail file, but cannot lose mail.

Copying mail from an inbox in the system's mailer directory actually puts it in an intermediate file '`~/.newmail`'. This is because the interlocking is done by a C program that copies to another file. '`~/.newmail`' is deleted after mail merging is successful. If there is a crash at the wrong time, this file continues to exist and will be used as an inbox the next time you get new mail.

27.5 Multiple Mail Files

Rmail operates by default on your *primary mail file*, which is named '`~/RMAIL`' and receives your incoming mail from your system inbox file. But you can also have other mail files and edit them with Rmail. These files can receive mail through their own inboxes, or you can move messages into them with explicit Rmail commands (see Section 27.6 [Rmail Output], page 232).

`i` *file* `RET` Read *file* into Emacs and run Rmail on it (`rmail-input`).

`M-x set-rmail-inbox-list RET` *files* `RET`
: Specify inbox file names for current Rmail file to get mail from.

`g` Merge new mail from current Rmail file's inboxes (`rmail-get-new-mail`).

`C-u g` *file* `RET`
: Merge new mail from inbox file *file*.

To run Rmail on a file other than your primary mail file, you may use the `i` (`rmail-input`) command in Rmail. This visits the file, puts it in Rmail mode, and then gets new mail from the file's inboxes if any. The file should be in Rmail format. You can use `M-x rmail-input` even when not in Rmail.

The file you read with `i` should usually be a valid Rmail file. If it is not, then Rmail tries to decompose it into a stream of messages in various known formats. If it succeeds, it converts the whole file to an Rmail file.

Each Rmail file can contain a list of inbox file names; you can specify this list with `M-x set-rmail-inbox-list RET` *files* `RET`. The argument can contain any number of file names, separated by commas. It can also be empty, which specifies that this file should have no inboxes. Once a list of inboxes is specified, the Rmail file remembers it permanently until it is explicitly changed.

If an Rmail file has inboxes, new mail is merged in from the inboxes when the Rmail file is brought into Rmail, and when you use the `g` (`rmail-get-new-mail`) command. If the Rmail file specifies no inboxes, then no new mail is merged in at these times. As a special exception, if your primary mail file does not specify any inbox files, it uses the standard system inbox.

To merge mail from a file that is not the usual inbox, give the `g` key a numeric argument, as in `C-u g`. Then it reads a file name and merges mail from that file. The inbox file is not deleted or changed in any way when `g` with an argument is used. This is, therefore, a general way of merging one file of messages into another.

27.6 Copying Messages Out to Files

`o` *file* `RET`
: Append a copy of the current message to the file *file*, writing it in Rmail file format by default (`rmail-output-to-rmail-file`).

`C-o` *file* `RET`
: Append a copy of the current message to the file *file*, writing it in Unix mail file format by default (`rmail-output`).

If an Rmail file has no inboxes, the only way messages get into it is by means of explicit `o` commands.

`o` (`rmail-output-to-rmail-file`) appends the current message in Rmail format to the end of the specified file. This is the best command to use to move messages between Rmail files. If the other Rmail file is currently visited, the copying is done into the other file's Emacs buffer instead. You should eventually save it on disk.

The `C-o` (`rmail-output`) command in Rmail appends a copy of the current message to a specified file, in inbox file format. This is useful for moving messages into files to be read by other mail processors that do not understand Rmail format.

The `o` and `C-o` commands are actually equivalent (aside from having separate file name defaults) when you specify an existing file; both commands check the file's contents to determine which format to use (Rmail or inbox).

But it does matter which command you use when you specify a nonexistent file; `o` starts it in Rmail format, while `C-o` starts it in inbox format.

Copying a message with `o` or `C-o` gives the original copy of the message the '`filed`' attribute, so that '`filed`' appears in the mode line when such a message is current. If you like to keep just a single copy of every mail message, set the variable `rmail-delete-after-output` to `t`; then the `o` and `C-o` commands delete the original message after copying it. (You can undelete the original afterward.)

Copying messages in Unix format copies the header fields that are displayed in Rmail at the time. Thus, if you use the `t` command to view the entire header and then copy the message, the entire header is copied.

The variable `rmail-output-file-alist` lets you specify intelligent defaults for the output file, based on the contents of the current message. The value should be a list whose elements have this form:

(*regexp* . *filename*)

If there's a match for *regexp* in the current message, then the default file name for output is *filename*. If multiple elements match the message, the first matching element decides the default file name.

27.7 Labels

Each message can have various *labels* assigned to it as a means of classification. A label has a name; different names mean different labels. Any given label is either present or absent on a particular message. A few label names have standard meanings and are given to messages automatically by Rmail when appropriate; these special labels are called *attributes*. All other labels are assigned only by users.

`a` *label* `RET`
: Assign the label *label* to the current message (`rmail-add-label`).

`k` *label* `RET`
: Remove the label *label* from the current message (`rmail-kill-label`).

`C-M-n` *labels* `RET`
: Move to the next message that has one of the labels *labels* (`rmail-next-labeled-message`).

`C-M-p` *labels* `RET`
: Move to the previous message that has one of the labels *labels* (`rmail-previous-labeled-message`).

`C-M-l` *labels* `RET`
: Make a summary of all messages containing any of the labels *labels* (`rmail-summary-by-labels`).

Specifying an empty string for one these commands means to use the last label specified for any of these commands.

The `a` (`rmail-add-label`) and `k` (`rmail-kill-label`) commands allow you to assign or remove any label on the current message. If the *label* argument is empty, it means to assign or remove the same label most recently assigned or removed.

Once you have given messages labels to classify them as you wish, there are two ways to use the labels: in moving and in summaries.

The command `C-M-n` *labels* `RET` (`rmail-next-labeled-message`) moves to the next message that has one of the labels *labels*. The argument *labels* specifies one or more label names, separated by commas. `C-M-p` (`rmail-previous-labeled-message`) is similar, but moves backwards to previous messages. A preceding numeric argument to either one serves as a repeat count.

The command `C-M-l` *labels* `RET` (`rmail-summary-by-labels`) displays a summary containing only the messages that have at least one of a specified set of messages. The argument *labels* is one or more label names, separated by commas. See Section 27.9 [Rmail Summary], page 237, for information on summaries.

If the *labels* argument to `C-M-n`, `C-M-p` or `C-M-l` is empty, it means to use the last set of labels specified for any of these commands.

Some labels such as '`deleted`' and '`filed`' have built-in meanings and are assigned to or removed from messages automatically at appropriate times; these labels are called *attributes*. Here is a list of Rmail attributes:

'`unseen`'
: Means the message has never been current. Assigned to messages when they come from an inbox file, and removed when a message is made current. When you start Rmail, it initially shows the first message that has this attribute.

'`deleted`'
: Means the message is deleted. Assigned by deletion commands and removed by undeletion commands (see Section 27.3 [Rmail Deletion], page 229).

'`filed`'
: Means the message has been copied to some other file. Assigned by the file output commands (see Section 27.5 [Rmail Files], page 231).

'`answered`'
: Means you have mailed an answer to the message. Assigned by the `r` command (`rmail-reply`). See Section 27.8 [Rmail Reply], page 235.

'forwarded'
: Means you have forwarded the message to other users. Assigned by the f command (rmail-forward). See Section 27.8 [Rmail Reply], page 235.

'edited'
: Means you have edited the text of the message within Rmail. See Section 27.10 [Rmail Editing], page 239.

'resent'
: Means you have resent the message to other users. Assigned by the command M-x rmail-resend. See Section 27.8 [Rmail Reply], page 235.

All other labels are assigned or removed only by the user, and have no standard meaning.

27.8 Sending Replies

Rmail has several commands that use Mail mode to send outgoing mail. See Chapter 26 [Sending Mail], page 221, for information on using Mail mode. What are documented here are the special commands of Rmail for entering Mail mode. Note that the usual keys for sending mail, C-x m and C-x 4 m, are available in Rmail mode and work just as they usually do.

m
: Send a message (rmail-mail).

c
: Continue editing already started outgoing message (rmail-continue).

r
: Send a reply to the current Rmail message (rmail-reply).

f
: Forward current message to other users (rmail-forward).

C-u f
: Resend the current message to other users (rmail-resend).

M-m
: Try sending a bounced message a second time (rmail-retry-failure).

The most common reason to send a message while in Rmail is to reply to the message you are reading. To do this, type r (rmail-reply). This displays the '*mail*' buffer in another window, much like C-x 4 m, but preinitializes the 'Subject', 'To', 'CC' and 'In-reply-to' header fields based on the message you are replying to. The 'To' field is starts out as the address of the person who sent the message you received, and the 'CC' field starts out with all the other recipients of that message.

You can exclude certain recipients from being placed automatically in the 'CC', using the variable rmail-dont-reply-to-names. Its value should be a regular expression (as a string); any recipient that the regular expression matches, is excluded from the 'CC' field. The default value matches your own name, and any name starting with 'info-'. (Those names are excluded because there is a convention of using them for large mailing lists to broadcast announcements.)

To omit the 'CC' field completely for a particular reply, enter the reply command with a prefix argument: `C-u r`.

Once the '*mail*' buffer has been initialized, editing and sending the mail goes as usual (see Chapter 26 [Sending Mail], page 221). You can edit the presupplied header fields if they are not right for you. You can also use the commands of Mail mode, including `C-c C-y` to yank in the message that you are replying to, and `C-c C-q` to fill what was thus yanked. You can also switch to the Rmail buffer, select a different message, switch back, and yank the new current message.

Sometimes a message does not reach its destination. Mailers usually send the failed message back to you, enclosed in a *failure message*. The Rmail command `M-m` (`rmail-retry-failure`) prepares to send the same message a second time: it sets up a '*mail*' buffer with the same text and header fields as before. If you type `C-c C-c` right away, you send the message again exactly the same as the first time. Alternatively, you can edit the text or headers and then send it.

Another frequent reason to send mail in Rmail is to *forward* the current message to other users. `f` (`rmail-forward`) makes this easy by preinitializing the '*mail*' buffer with the current message as the text, and a subject designating a forwarded message. All you have to do is fill in the recipients and send. When you forward a message, recipients get a message which is "from" you, and which has the original message in its contents.

Resending is an alternative similar to forwarding; the difference is that resending sends a message that is "from" the original sender, just as it reached you—with a few added header fields 'Resent-from' and 'Resent-to' to indicate that it came via you. To resend a message in Rmail, use `C-u f`. (`f` runs `rmail-forward`, which is programmed to invoke `rmail-resend` if you provide a numeric argument.)

The `m` (`rmail-mail`) command is used to start editing an outgoing message that is not a reply. It leaves the header fields empty. Its only difference from `C-x 4 m` is that it makes the Rmail buffer accessible for `C-c C-y`, just as `r` does. Thus, `m` can be used to reply to or forward a message; it can do anything `r` or `f` can do.

The `c` (`rmail-continue`) command resumes editing the '*mail*' buffer, to finish editing an outgoing message you were already composing, or to alter a message you have sent.

If you set the variable `rmail-mail-new-frame` to a non-`nil` value, then all the Rmail commands to start sending a message create a new frame to edit it in. This frame is deleted when you send the message, or when you use the '`Don't Send`' item in the '`Mail`' menu.

27.9 Summaries

A *summary* is a buffer containing one line per message to give you an overview of the mail in an Rmail file. Each line shows the message number, the sender, the labels, and the subject. When the summary buffer is selected, you can use almost all Rmail commands; these apply to the message described by the current line of the summary. Moving point in the summary buffer selects messages as you move to their summary lines.

A summary buffer applies to a single Rmail file only; if you are editing multiple Rmail files, each one has its own summary buffer (if you have asked for one). The summary buffer name is made by appending '`-summary`' to the Rmail buffer's name. Normally only one summary buffer is displayed at a time.

27.9.1 Making Summaries

Here are the commands to create a summary for the current Rmail file. Once the Rmail file has a summary buffer, changes in the Rmail file (such as deleting or expunging messages, and getting new mail) automatically update the summary.

`h`
`C-M-h` Summarize all messages (`rmail-summary`).

`l` *labels* `RET`
`C-M-l` *labels* `RET`
: Summarize message that have one or more of the specified labels (`rmail-summary-by-labels`).

`C-M-r` *rcpts* `RET`
: Summarize messages that have one or more of the specified recipients (`rmail-summary-by-recipients`).

`C-M-t` *topic* `RET`
: Summarize messages that have a match for the specified regexp *topic* in their subjects (`rmail-summary-by-topic`).

The `h` or `C-M-h` (`rmail-summary`) command fills the summary buffer for the current Rmail file with a summary of all the messages in the file. It then displays and selects the summary buffer in another window.

`C-M-l` *labels* `RET` (`rmail-summary-by-labels`) makes a partial summary mentioning only the messages that have one or more of the labels *labels*. *labels* should contain label names separated by commas.

`C-M-r` *rcpts* `RET` (`rmail-summary-by-recipients`) makes a partial summary mentioning only the messages that have one or more of the recipients *rcpts*. *rcpts* should contain mailing addresses separated by commas.

`C-M-t` *topic* `RET` (`rmail-summary-by-topic`) makes a partial summary mentioning only the messages whose subjects have a match for the regular expression *topic*.

Note that there is only one summary buffer for any Rmail file; making one kind of summary discards any previously made summary.

27.9.2 Editing in Summaries

You can use the Rmail summary buffer to do almost anything you can do in the Rmail buffer itself. In fact, once you have a summary buffer, there's no need to switch back to the Rmail buffer.

You can select and display various messages in the Rmail buffer, from the summary buffer, just by moving point in the summary buffer to different lines. It doesn't matter what Emacs command you use to move point; whichever line point is on at the end of the command, that message is selected in the Rmail buffer.

Almost all Rmail commands work in the summary buffer as well as in the Rmail buffer. Thus, `d` in the summary buffer deletes the current message, `u` undeletes, and `x` expunges. `o` and `C-o` output the current message to a file; `r` starts a reply to it. You can scroll the current message while remaining in the summary buffer using `SPC` and `DEL`.

The Rmail commands to move between messages also work in the summary buffer, but with a twist: they move through the set of messages included in the summary. They also redisplay the Rmail buffer on the screen (unlike cursor motion commands, which update the contents of the Rmail buffer but don't display it in a window unless it already appears). Here is a list of these commands:

`n` Move to next line, skipping lines saying 'deleted', and select its message.

`p` Move to previous line, skipping lines saying 'deleted', and select its message.

`M-n` Move to next line and select its message.

`M-p` Move to previous line and select its message.

`>` Move to the last line, and select its message.

`<` Move to the first line, and select its message.

`M-s` *pattern* `RET`
: Search through messages for *pattern* starting with the current message; select the message found, and move point in the summary buffer to that message's line.

Deletion, undeletion, and getting new mail, and even selection of a different message all update the summary buffer when you do them in the Rmail buffer. If the variable `rmail-redisplay-summary` is non-`nil`, these actions also bring the summary buffer back onto the screen.

When you are finished using the summary, type `w` (`rmail-summary-wipe`) to kill the summary buffer's window. You can also exit Rmail while in the summary. `q` (`rmail-summary-quit`) kills the summary window, then saves the Rmail file and switches to another buffer.

27.10 Editing Within a Message

Rmail mode provides a few special commands for moving within and editing the current message. In addition, the usual Emacs commands are available (except for a few, such as `C-M-n` and `C-M-h`, that are redefined by Rmail for other purposes). However, the Rmail buffer is normally read-only, and to alter it you must use the Rmail command `e` described below.

`t`
: Toggle display of original headers (`rmail-toggle-headers`).

`e`
: Edit current message (`rmail-edit-current-message`).

Rmail reformats the header of each message before displaying it. Normally this involves deleting most header fields, on the grounds that they are not interesting. The variable `rmail-ignored-headers` specifies a regexp that matches the header fields to discard in this way. The original headers are saved permanently, and to see what they look like, use the `t` command (`rmail-toggle-headers`). This discards the reformatted headers of the current message and displays it with the original headers. Repeating `t` reformats the message again. Selecting the message again also reformats.

The Rmail buffer is normally read only, and most of the characters you would type to modify it (including most letters) are redefined as Rmail commands. This is usually not a problem since it is rare to want to change the text of a message. When you do want to do this, the way is to type `e` (`rmail-edit-current-message`), which changes from Rmail mode into Rmail Edit mode, another major mode which is nearly the same as Text mode. The mode line indicates this change.

In Rmail Edit mode, letters insert themselves as usual and the Rmail commands are not available. When you are finished editing the message and are ready to go back to Rmail, type `C-c C-c`, which switches back to Rmail mode. Alternatively, you can return to Rmail mode but cancel all the editing that you have done by typing `C-c C-]`.

Entering Rmail Edit mode runs the hook `text-mode-hook`; then it runs the hook `rmail-edit-mode-hook` (see Section 31.2.3 [Hooks], page 294). It adds the attribute '`edited`' to the message.

27.11 Digest Messages

A *digest message* is a message which exists to contain and carry several other messages. Digests are used on some moderated mailing lists; all the messages that arrive for the list during a period of time such as one day are put inside a single digest which is then sent to the subscribers. Transmitting the single digest uses much less computer time than transmitting the individual messages even though the total size is the same, because the per-message overhead in network mail transmission is considerable.

When you receive a digest message, the most convenient way to read it is to *undigestify* it: to turn it back into many individual messages. Then you can read and delete the individual messages as it suits you.

To do this, type `M-x undigestify-rmail-message` after selecting the digest message. This extracts the submessages as separate Rmail messages, and inserts them following the digest. The digest message itself is flagged as deleted.

27.12 Converting an Rmail File to Mailbox Format

The command `M-x unrmail` converts a file in Rmail format to ordinary system mailbox format, so that you can use it with other mail-editing tools. You must specify two arguments, the name of the Rmail file and the name to use for the converted file. `M-x unrmail` does not alter the Rmail file itself.

27.13 Reading Rot13 Messages

Mailing list messages that might offend some readers are sometimes encoded in a simple code called *rot13*—so named because it rotates the alphabet by 13 letters. This code is not for secrecy, as it provides none; rather, it enables those who might be offended to avoid ever seeing the real text of the message.

To view a buffer using the rot13 code, use the command `M-x rot13-other-window`. This displays the current buffer in another window which applies the code when displaying the text.

28 Dired, the Directory Editor

Dired makes an Emacs buffer containing a listing of a directory, and optionally some of its subdirectories as well. You can use the normal Emacs commands to move around in this buffer, and special Dired commands to operate on the files listed.

28.1 Entering Dired

To invoke Dired, do `C-x d` or `M-x dired`. The command reads a directory name or wildcard file name pattern as a minibuffer argument to specify which files to list. Where `dired` differs from `list-directory` is in putting the buffer into Dired mode so that the special commands of Dired are available.

The variable `dired-listing-switches` specifies the options to give to `ls` for listing directory; this string *must* contain '`-l`'. If you use a numeric prefix argument with the `dired` command, you can specify the `ls` switches with the minibuffer after you finish entering the directory specification.

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

28.2 Commands in the Dired Buffer

The Dired buffer is "read-only", and inserting text in it is not useful, so ordinary printing characters such as `d` and `x` are used for special Dired commands. Some Dired commands *mark* or *flag* the *current file* (that is, the file on the current line); other commands operate on the marked files or on the flagged files.

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

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

28.3 Deleting Files with Dired

The primary use of Dired is to *flag* files for deletion and then delete the files previously flagged.

`d` Flag this file for deletion.

`u` Remove deletion flag on this line.

`DEL` Move point to previous line and remove the deletion flag on that line.

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

`#` Flag all auto-save files (files whose names start and end with '`#`') for deletion (see Section 15.5 [Auto Save], page 103).

`~` Flag all backup files (files whose names end with '`~`') for deletion (see Section 15.3.1 [Backup], page 99).

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

`% d` *regexp* `RET`
: Flag for deletion all files whose names match the regular expression *regexp* (`dired-flag-files-regexp`). This is just like `% m` except that it uses '`D`' instead of '`*`' to mark the files.

  Only the non-directory part of the file name is used in matching. Use '`^`' and '`$`' to anchor matches. Exclude subdirectories by hiding them (see Section 28.12 [Hiding Subdirectories], page 249).

You can flag a file for deletion by moving to the line describing the file and typing `d`. The deletion flag is visible as a '`D`' at the beginning of the line. This command moves point to the beginning of the next line, so that repeated `d` commands flag successive files.

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

To delete the flagged files, type `x` (`dired-expunge`). This command first displays a list of all the file names flagged for deletion, and requests confirmation with `yes`. Once you confirm, `x` deletes all the flagged files, then deletes their lines from the text of the Dired buffer. The shortened Dired buffer remains selected.

If you answer `no` or quit with `C-g` when asked to confirm, you return immediately to Dired, with the deletion flags still present in the buffer, and no files actually deleted.

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

`#` flags for deletion all files whose names look like auto-save files (see Section 15.5 [Auto Save], page 103)—that is, files whose names begin and end with '`#`'. `~` flags for deletion all files whose names say they are backup files (see Section 15.3.1 [Backup], page 99)—that is, whose names end in '`~`'.

`.` (Period) flags just some of the backup files for deletion: all but the oldest few and newest few backups of any one file. Normally `dired-kept-versions` (**not** `kept-new-versions`; that applies only when saving) specifies the number of newest versions of each file to keep, and `kept-old-versions` specifies the number of oldest versions to keep.

Period with a positive numeric argument, as in `C-u 3 .`, specifies the number of newest versions to keep, overriding `dired-kept-versions`. A negative numeric argument overrides `kept-old-versions`, using minus the value of the argument to specify the number of oldest versions of each file to keep.

The `% d` command flags all files whose names match a specified regular expression (`dired-flag-files-regexp`). Only the non-directory part of the file name is used in matching. You can use '`^`' and '`$`' to anchor matches. You can exclude subdirectories by hiding them (see Section 28.12 [Hiding Subdirectories], page 249).

28.4 Visiting Files in Dired

There are several Dired commands for visiting or examining the files listed in the Dired buffer. All of them apply to the current line's file; if that file is really a directory, these commands invoke Dired on that subdirectory (using a separate Dired buffer).

`f`
: Visit the file described on the current line, like typing `C-x C-f` and supplying that file name (`dired-find-file`). See Section 15.2 [Visiting], page 94.

`o`
: Like `f`, but uses another window to display the file's buffer (`dired-find-file-other-window`). The Dired buffer remains visible in the first window. This is like using `C-x 4 C-f` to visit the file. See Chapter 17 [Windows], page 125.

`C-o`
: Visit the file described on the current line, and display the buffer in another window, but do not select that window (`dired-display-file`).

`v`
: View the file described on the current line, using `M-x view-file` (`dired-view-file`).

Viewing a file is like visiting it, but is slanted toward moving around in the file conveniently and does not allow changing the file. See Section 15.10 [Misc File Ops], page 116.

28.5 Dired Marks vs. Flags

Instead of flagging a file with 'D', you can *mark* the file with some other character (usually '*'). Most Dired commands other than "expunge" (`x`) operate on files marked with '*'.

Here are some commands for marking with '*' (and also for unmarking). (See Section 28.3 [Dired Deletion], page 241, for commands to flag and unflag files.)

`m` Mark the current file with '*' (`dired-mark`). With a numeric argument *n*, mark the next *n* files starting with the current file. (If *n* is negative, mark the previous $-n$ files.)

`*` Mark all executable files with '*' (`dired-mark-executables`). With a prefix argument, unmark all those files.

`@` Mark all symbolic links with '*' (`dired-mark-symlinks`). With a prefix argument, unmark all those files.

`/` Mark with '*' all files which are actually directories, except for '.' and '..' (`dired-mark-directories`). With a prefix argument, unmark all those files.

`M-DEL` *markchar*
: Remove all marks that use the character *markchar* (`dired-unmark-all-files`). If you specify `RET` as *markchar*, this command removes all marks, no matter what the marker character is.

  With a numeric argument, this command queries about each marked file, asking whether to remove its mark. You can answer `y` meaning yes, `n` meaning no, `!` to remove the marks from the remaining files without asking about them.

`c` *old new* Replace all marks that use the character *old* with marks that use the character *new* (`dired-change-marks`). This command is the only way to create or use marks other than '*' or 'D'. The arguments are single characters—do not use `RET` to terminate them.

  You can use almost any character as a mark character by means of this command, to distinguish various classes of files. If *old* is a space (' '), then the command operates on all unmarked files; if *new* is a space, then the command unmarks the files it acts on.

To illustrate the power of this command, here is how to put '*' marks on all the files that are unmarked, while unmarking all those that have '*' marks:

```
c * t  c SPC *  c t SPC
```

`% m` *regexp* `RET`
: Mark (with '*') all files whose names match the regular expression *regexp* (`dired-mark-files-regexp`).

  Only the non-directory part of the file name is used in matching. Use '^' and '$' to anchor matches. Exclude subdirectories by hiding them (see Section 28.12 [Hiding Subdirectories], page 249).

28.6 Operating on Files

This section describes the basic Dired commands to operate on one file or several files. All of these commands are capital letters; all of them use the minibuffer, either to read an argument or to ask for confirmation, before they act. All use the following convention to decide which files to manipulate:

- If you give the command a numeric prefix argument *n*, it operates on the next *n* files, starting with the current file. (If *n* is negative, the command operates on the $-n$ files preceding the current line.)
- Otherwise, if some files are marked with '*', the command operates on all those files.
- Otherwise, the command operates on the current file only.

Here are the commands that operate on files in this way:

`C` *new* `RET`
: Copy the specified files (`dired-do-copy`). The argument *new* is the directory to copy into, or (if copying a single file) the new name.

  If `dired-copy-preserve-time` is non-`nil`, then copying with this command sets the modification time of the new file to be the same as that of the old file.

`R` *new* `RET`
: Rename the specified files (`dired-do-rename`). The argument *new* is the directory to rename into, or (if renaming a single file) the new name.

  Dired automatically changes the visited file name of buffers associated with renamed files so that they refer to the new names.

`H` *new* `RET`
: Make hard links to the specified files (`dired-do-hardlink`). The argument *new* is the directory to make the links in, or (if making just one link) the name to give the link.

`S` *new* `RET` Make symbolic links to the specified files (`dired-do-symlink`). The argument *new* is the directory to make the links in, or (if making just one link) the name to give the link.

`M` *modespec* `RET`
: Change the mode (also called "permission bits") of the specified files (`dired-do-chmod`). This calls the `chmod` program, so *modespec* can be any argument that `chmod` can handle.

`G` *newgroup* `RET`
: Change the group of the specified files to *newgroup* (`dired-do-chgrp`).

`O` *newowner* `RET`
: Change the owner of the specified files to *newowner* (`dired-do-chown`). (On most systems, only the superuser can do this.)

  The variable `dired-chown-program` specifies the name of the program to use to do the work (different systems put `chown` in different places).

`P` *command* `RET`
: Print the specified files (`dired-do-print`). You must specify the command to print them with, but the minibuffer starts out with a suitable guess made using the variables `lpr-command` and `lpr-switches` (the same variables that `lpr-file` uses; see Section 30.5 [Hardcopy], page 283).

`Z` Compress or uncompress the specified files (`dired-do-compress`). If the file appears to be a compressed file, it is uncompressed; otherwise, it is compressed.

`L` Load the specified Emacs Lisp files (`dired-do-load`). See Section 23.4 [Lisp Libraries], page 205.

`B` Byte compile the specified Emacs Lisp files (`dired-do-byte-compile`). See section "Byte Compilation" in *The Emacs Lisp Reference Manual*.

28.7 Shell Commands in Dired

The dired command `!` (`dired-do-shell-command`) reads a shell command string in the minibuffer and runs the shell command on all the specified files. There are two ways of applying a shell command to multiple files:

- If you use '`*`' in the shell command, then it runs just once, with the list of file names substituted for the '`*`'. The order of file names is the order of appearance in the Dired buffer.

  Thus, `! tar cf foo.tar * RET` runs `tar` on the entire list of file names, putting them into one tar file '`foo.tar`'.

- If the command string doesn't contain '`*`', then it runs once *for each file*, with the file name added at the end.

  For example, `! uudecode RET` runs `uudecode` on each file.

What if you want to run the shell command once for each file but with the file name inserted in the middle? Or if you want to use the file names in a more complicated fashion? Use a shell loop. For example, this shell command would run `uuencode` on each of the specified files, writing the output into a corresponding '`.uu`' file:

```
for file in *; uuencode $file $file >$file.uu; done
```

The working directory for the shell command is the top level directory of the Dired buffer.

The `!` command does not attempt to update the Dired buffer to show new or modified files, because it doesn't really understand shell commands, and does not know what files the shell command changed. Use the `g` command to update the Dired buffer (see Section 28.13 [Dired Updating], page 250).

28.8 Transforming File Names in Dired

Here are commands that alter file names in a systematic way:

`% u`
: Rename each of the selected files to an upper case name (`dired-upcase`). If the old file names are '`Foo`' and '`bar`', the new names are '`FOO`' and '`BAR`'.

`% l`
: Rename each of the selected files to a lower case name (`dired-downcase`). If the old file names are '`Foo`' and '`bar`', the new names are '`foo`' and '`bar`'.

`% R` *from* `RET` *to* `RET`
`% C` *from* `RET` *to* `RET`
`% H` *from* `RET` *to* `RET`
`% S` *from* `RET` *to* `RET`
: These four commands rename, copy, make hard links and make soft links, in each case computing the new name by regular expression substitution from the name of the old file.

The four regular expression substitution commands effectively perform `query-replace-regexp` on the selected file names in the Dired buffer. They read two arguments: a regular expression *from*, and a substitution pattern *to*.

The commands match each "old" file name against the regular expression *from*, and then replace the matching part with *to*. You can use '`\&`' and '`\`*digit*' in *to* to refer to all or part of the old file name, as in `query-replace` (see Section 13.7.4 [Query Replace], page 86).

For example, `% R ^.*$ RET x-\& RET` renames each selected file by prepending '`x-`' to its name. The inverse of this, removing '`x-`' from the front of each file name, is also possible: one method is `% R ^x-.*$ RET \& RET`; another is `% R ^x- RET RET`. (Use '`^`' and '`$`' to anchor matches that should span the whole filename.)

If the regular expression matches more than once in a file name, only the first match is replaced.

Normally, the replacement process does not consider the files' directory names; it operates on the file name within the directory. If you specify a prefix argument of zero, then replacement affects the entire absolute file name including directory name.

Often you will want to apply the command to all files matching the same *regexp* that you use in the command. To do this, mark those files with `% m` *regexp* `RET`, then use the same regular expression in the command to operate on the files. To make this easier, the commands to operate use the last regular expression specified in a `%` command as a default.

28.9 File Comparison with Dired

Here are two Dired commands that compare specified files using `diff`.

`=` Compare the current file (the file at point) with another file (the file at the mark) using the `diff` program (`dired-diff`). The file at the mark is the first argument of `diff`, and the file at point is the second argument.

`M-=` Compare the current file with its latest backup file (`dired-backup-diff`). If the current file is itself a backup, compare it with the file it is a backup of; this way, you can compare a file with any backup version of your choice.

The backup file is the first file given to `diff`.

28.10 Subdirectories in Dired

A Dired buffer displays just one directory in the normal case; but you can optionally include its subdirectories as well.

The simplest way to include multiple directories in one Dired buffer is to specify the options '`-lR`' for running `ls`. (If you give a numeric argument when you run Dired, then you can specify these options in the minibuffer.) That produces a recursive directory listing showing all subdirectories at all levels.

But usually all the subdirectories are too many; usually you will prefer to include specific subdirectories only. You can do this with the `i` command:

`i` Insert the contents of a subdirectory later in the buffer.

Use the `i` (`dired-maybe-insert-subdir`) command on a line that describes a file which is a directory. It inserts the contents of that directory into the same Dired buffer, and moves there. Inserted subdirectory contents follow the top-level directory of the Dired buffer, just as they do in '`ls -lR`' output.

If the subdirectory's contents are already present in the buffer, the `i` command just moves to it.

In either case, `i` sets the Emacs mark before moving, so `C-x C-x` takes you back to the old position in the buffer (the line describing that subdirectory).

Use the `l` command (`dired-do-redisplay`) to update the subdirectory's contents. Use `k` to delete the subdirectory. See Section 28.13 [Dired Updating], page 250.

28.11 Moving Over Subdirectories

When a Dired buffer lists subdirectories, you can use the page motion commands `C-x [` and `C-x ]` to move by entire directories.

The following commands move across, up and down in the tree of directories within one Dired buffer. They move to *directory header lines*, which are the lines that give a directory's name, at the beginning of the directory's contents.

`C-M-n` Go to next subdirectory header line, regardless of level (`dired-next-subdir`).

`C-M-p` Go to previous subdirectory header line, regardless of level (`dired-prev-subdir`).

`C-M-u` Go up to the parent directory's header line (`dired-tree-up`).

`C-M-d` Go down in the directory tree, to the first subdirectory's header line (`dired-tree-down`).

28.12 Hiding Subdirectories

Hiding a subdirectory means to make it invisible, except for its header line, via selective display (see Section 12.3 [Selective Display], page 71).

`$` Hide or reveal the subdirectory that point is in, and move point to the next subdirectory (`dired-hide-subdir`). A numeric argument serves as a repeat count.

`M-$` Hide all subdirectories in this Dired buffer, leaving only their header lines (`dired-hide-all`). Or, if any subdirectory is currently hidden, make all subdirectories visible again. You can use this command to get an overview in very deep directory trees or to move quickly to subdirectories far away.

Ordinary Dired commands never consider files inside a hidden subdirectory. For example, the commands to operate on marked files ignore files in hidden directories even if they are marked. Thus you can use hiding to temporarily exclude subdirectories from operations without having to remove the markers.

The subdirectory hiding commands toggle; that is they unhide what was hidden and vice versa.

28.13 Updating the Dired Buffer

This section describes commands to update the Dired buffer to reflect outside (non-Dired) changes in the directories and files, and to delete part of the Dired buffer.

`g` Update the entire contents of the Dired buffer (`revert-buffer`).

`l` Update the specified files (`dired-do-redisplay`).

`k` Delete the specified *file lines*—not the files, just the lines (`dired-do-kill-lines`).

Type `g` (`revert-buffer`) to update the contents of the Dired buffer, based on changes in the files and directories listed. This preserves all marks except for those on files that have vanished. Hidden subdirectories are updated but remain hidden.

To update only some of the files, type `l` (`dired-do-redisplay`). This command applies to the next *n* files, or to the marked files if any, or to the current file. Updating them means reading their current status from the file system and changing the buffer to reflect it properly.

If you use `l` on a subdirectory header line, it updates the contents of the corresponding subdirectory.

To delete the specified *file lines*—not the files, just the lines—type `k` (`dired-do-kill-lines`). This command applies to the next *n* files, or to the marked files if any, or to the current file.

If you kill the line for a file that is a directory, the directory's contents are also deleted from the buffer. Typing `C-u k` on the header line for a subdirectory is another way to delete a subdirectory from the Dired buffer.

The `g` command brings back any individual lines that you have killed in this way, but not subdirectories—you must use `i` to reinsert each subdirectory.

28.14 Dired and `find`

You can select a set of files for display in a Dired buffer more flexibly by using the `find` utility to choose the files.

To search for files with names matching a wildcard pattern use `M-x find-name-dired`. It reads arguments *directory* and *pattern*, and chooses all the files in *directory* or its subdirectories whose individual names match *pattern*.

The files thus chosen are displayed in a Dired buffer in which the ordinary Dired commands are available.

If you want to test the contents of files, rather than their names, use `M-x find-grep-dired`. This command reads two minibuffer arguments, *directory* and *regexp*; it chooses all the files in *directory* or its subdirectories that contain a match for *regexp*. It works by running the programs `find` and `grep`.

The most general command in this series is `M-x find-dired`, which lets you specify any condition that `find` can test. It takes two minibuffer arguments, *directory* and *find-args*; it runs `find` in *directory*, passing *find-args* to tell `find` what condition to test. To use this command, you need to know how to use `find`.

29 The Calendar and the Diary

Emacs provides the functions of a desk calendar, with a diary of planned or past events. To enter the calendar, type `M-x calendar`; this displays a three-month calendar centered on the current month, with point on the current date. If you use prefix argument as in `C-u M-x calendar`, it prompts you for the month and year to be the center of the three-month calendar. The calendar uses its own buffer and the major mode is Calendar mode.

To exit the calendar, type `q`.

29.1 Movement in the Calendar

Calendar mode lets you move in logical units of time such as days, weeks, months, and years. If you move outside the three months originally displayed, the calendar display scrolls automatically through time. Moving to a date lets you view its holidays or diary entries, convert it to other calendars; moving longer time periods is useful simply to scroll the calendar.

29.1.1 Motion by Integral Days, Weeks, Months, Years

The commands for movement in the calendar buffer parallel the commands for movement in text. You can move forward and backward by days, weeks, months, and years.

`C-f` Move point one day forward (`calendar-forward-day`).

`C-b` Move point one day backward (`calendar-backward-day`).

`C-n` Move point one week forward (`calendar-forward-week`).

`C-p` Move point one week backward (`calendar-backward-week`).

`M-}` Move point one month forward (`calendar-forward-month`).

`M-{` Move point one month backward (`calendar-backward-month`).

`C-x ]` Move point one year forward (`calendar-forward-year`).

`C-x [` Move point one year backward (`calendar-forward-year`).

The day and week commands are natural analogues of the usual Emacs commands for moving by characters and by lines. Just as `C-n` usually moves to the same column in the following line, in Calendar mode it moves to the same day in the following week. And `C-p` moves to the same day in the previous week.

The arrow keys are equivalent to `C-f`, `C-b`, `C-n` and `C-p`, just as they normally are in other modes.

The commands for motion by months and years work like those for weeks, but move a larger distance. The month commands `M-}` and `M-{` move forward or backward by an entire month's time. The year commands `C-x ]` and `C-x [` move forward or backward a whole year.

The easiest way to remember these commands is to consider months and years analogous to paragraphs and pages of text, respectively. But the commands themselves are not quite analogous. The ordinary Emacs paragraph commands move to the beginning or end of a paragraph, whereas these month and year commands move by an entire month or an entire year, which usually involves skipping across the end of a month or year.

All these commands accept a numeric argument as a repeat count. For convenience, the digit keys and the minus sign specify numeric arguments in Calendar mode even without the Meta modifier. For example, `100 C-f` moves point 100 days forward from its present location.

29.1.2 Beginning or End of Week, Month or Year

A week (or month, or year) is not just a quantity of days; we think of new weeks (months, years) as starting on particular days. So Calendar mode provides commands to move to the beginning or end of a week, month or year:

`C-a` Move point to beginning of week (`calendar-beginning-of-week`).

`C-e` Move point to end of week (`calendar-end-of-week`).

`M-a` Move point to beginning of month (`calendar-beginning-of-month`).

`M-e` Move point to end of month (`calendar-end-of-month`).

`M-<` Move point to beginning of year (`calendar-beginning-of-year`).

`M->` Move point to end of year (`calendar-end-of-year`).

These commands also take numeric arguments as repeat counts, with the repeat count indicating how many weeks, months, or years to move backward or forward.

By default, weeks begin on Sunday. To make them begin on Monday instead, set the variable `calendar-week-start-day` to 1.

29.1.3 Particular Dates

Calendar mode provides commands for getting to a particular date specified absolutely.

`g d` Move point to specified date (`calendar-goto-date`).

`o` Center calendar around specified month (`calendar-other-month`).

`.` Move point to today's date (`calendar-current-month`).

`g d` (`calendar-goto-date`) prompts for a year, a month, and a day of the month, and then moves to that date. Because the calendar includes all dates from the beginning of the current era, you must type the year in its entirety; that is, type '`1990`', not '`90`'.

`o` (`calendar-other-month`) prompts for a month and year, then centers the three-month calendar around that month.

You can return to today's date with `.` (`calendar-current-month`).

29.2 Scrolling in the Calendar

The calendar display scrolls automatically through time when you move out of the visible portion. You can also scroll it manually. Imagine that the calendar window contains a long strip of paper with the months on it. Scrolling it means moving the strip so that new months become visible in the window.

`C-x <` Scroll calendar one month forward (`scroll-calendar-left`).

`C-x >` Scroll calendar one month backward (`scroll-calendar-right`).

`C-v`
`NEXT` Scroll calendar three months forward (`scroll-calendar-left-three-months`).

`M-v`
`PRIOR` Scroll calendar three months backward (`scroll-calendar-right-three-months`).

`SPC` Scroll the next window (`scroll-other-window`).

The most basic calendar scroll commands scroll by one month at a time. This means that there are two months of overlap between the display before the command and the display after. `C-x <` scrolls the calendar contents one month to the left; that is, it moves the display forward in time. `C-x >` scrolls the contents to the right, which moves backwards in time.

The commands `C-v` and `M-v` scroll the calendar by an entire "screenful"—three months—in analogy with the usual meaning of these commands. `C-v` makes later dates visible and `M-v` makes earlier dates visible. These commands take a numeric argument as a repeat count; in particular, since `C-u` (`universal-argument`) multiplies the next command by four, typing `C-u`

C-v scrolls the calendar forward by a year and typing C-u M-v scrolls the calendar backward by a year.

The function keys NEXT and PRIOR are equivalent to C-v and M-v, just as they are in other modes.

In Calendar mode, you can use SPC (scroll-other-window) to scroll the other window. This is handy when you display a list of holidays or diary entries in another window.

29.3 Counting Days

M-= Display the number of days in the current region (calendar-count-days-region).

To determine the number of days in the region, type M-= (calendar-count-days-region). The numbers of days printed is *inclusive*; that is, it includes the days specified by mark and point.

29.4 Miscellaneous Calendar Commands

p d Display day-in-year (calendar-print-day-of-year).

C-c C-l Regenerate the calendar window (redraw-calendar).

q Exit from calendar (exit-calendar).

To print the number of days elapsed since the start of the year, or the number of days remaining in the year, type the p d command (calendar-print-day-of-year). This displays both of those numbers in the echo area. The number of days elapsed includes the selected date. The number of days remaining does not include that date.

If the calendar window gets corrupted, type C-c C-l (redraw-calendar) to redraw it.

To exit from the calendar, type q (exit-calendar). This buries all buffers related to the calendar and returns the window display to what it was when you entered the calendar.

29.5 Holidays

The Emacs calendar knows about all major and many minor holidays, and can display them.

h Display holidays for the selected date (calendar-cursor-holidays).

`x` Mark holidays in the calendar window (`mark-calendar-holidays`).

`u` Unmark calendar window (`calendar-unmark`).

`a` List all holidays for the displayed three months in another window (`list-calendar-holidays`).

`M-x holidays`
List all holidays for three months around today's date in another window.

To see if any holidays fall on a given date, position point on that date in the calendar window and use the `h` command. This names the holidays for that date, in the echo area if they fit there, otherwise in a separate window.

To find the distribution of holidays for a wider period, use the `x` command. This displays a '*' after each date on which a holiday falls. The command applies both to the currently visible months and to other months that subsequently become visible by scrolling. To turn marking off and erase the current marks, type `u`, which also erases any diary marks (see Section 29.9 [Diary], page 263).

To get even more detailed information, use the `a` command, which displays a separate buffer containing a list of all holidays in the current three-month range.

The command `M-x holidays` displays the list of holidays for the current month and the preceding and succeeding months; this works even if you don't have a calendar window. If you want the list of holidays centered around a different month, use `C-u M-x holidays`, which prompts for the month and year.

The holidays known to Emacs include American holidays and the major Christian, Jewish, and Islamic holidays; when floating point is available, the calendar also knows about solstices and equinoxes.

The dates used by Emacs for holidays are based on *current practice*, not historical fact. Historically, for instance, the start of daylight savings time and even its existence have varied from year to year, but present American law mandates that daylight savings time begins on the first Sunday in April. In an American locale, Emacs always uses this definition, even though it is wrong for some prior years.

29.6 Times of Sunrise and Sunset

Emacs can tell you, to within a minute or two, the times of sunrise and sunset for any date, if floating point is available.

`S` Display times of sunrise and sunset for the selected date (`calendar-sunrise-sunset`).

`M-x sunrise-sunset`
: Display times of sunrise and sunset for today's date.

`C-u M-x sunrise-sunset`
: Display times of sunrise and sunset for a specified date.

Within the calendar, to display the *local times* of sunrise and sunset in the echo area, move point to the date you want, and type `S`.

The command `M-x sunrise-sunset` is available outside the calendar to print this information for today's date or a specified date. To specify a date other than today, use `C-u M-x sunrise-sunset`, which prompts for the year, month, and day.

Because the times of sunrise and sunset depend on the location on earth, you need to tell Emacs your latitude, longitude, and location name before using these commands. Here is an example of what to set:

```
(setq calendar-latitude 40.1)
(setq calendar-longitude -88.2)
(setq calendar-location-name "Urbana, IL")
```

Use one decimal place in the values of `calendar-latitude` and `calendar-longitude`.

Your time zone also affects the local time of sunrise and sunset. Emacs usually gets this information from the operating system, but if these values are not what you want (or if the operating system does not supply them), you must set them yourself. Here is an example:

```
(setq calendar-time-zone -360)
(setq calendar-standard-time-zone-name "CST")
(setq calendar-daylight-time-zone-name "CDT")
```

The value of `calendar-time-zone` is the number of minutes difference between your local standard time and Coordinated Universal Time (Greenwich time). The values of `calendar-standard-time-zone-name` and `calendar-daylight-time-zone-name` are the abbreviations used in your time zone.

Emacs displays the times of sunrise and sunset *corrected for daylight savings time*. See Chapter 30 [Daylight Savings], page 271, for how daylight savings time is determined.

You can display the times of sunrise and sunset for any location and any date with `C-u C-u M-x sunrise-sunset`. This asks you for a longitude, latitude, number of minutes difference from Coordinated Universal Time, and date, and then tells you the times of sunrise and sunset for that location on that date.

29.7 Phases of the Moon

Emacs can tell you the dates and times of the phases of the moon (new moon, first quarter, full moon, last quarter), if floating point is available. This feature is useful for debugging problems that "depend on the phase of the moon."

`M` Display the dates and times for all the quarters of the moon for the three-month period shown (`calendar-phases-of-moon`).

`M-x phases-of-moon`
: Display dates and times of the quarters of the moon for three months around today's date.

Within the calendar, use the `M` command to display a separate buffer of the phases of the moon for the current three-month range. The dates and times listed are accurate to within a few minutes.

Outside the calendar, use the command `M-x phases-of-moon` to display the list of the phases of the moon for the current month and the preceding and succeeding months. For information about a different month, use `C-u M-x phases-of-moon`, which prompts for the month and year.

The dates and times given for the phases of the moon are given in local time (corrected for daylight savings, when appropriate); but if the variable `calendar-time-zone` is void, Coordinated Universal Time (the Greenwich time zone) is used. See Chapter 30 [Daylight Savings], page 271.

29.8 Conversion To and From Other Calendars

The Emacs calendar displayed is *always* the Gregorian calendar, sometimes called the "new style" calendar, which is used in most of the world today. However, this calendar did not exist before the sixteenth century and was not widely used before the eighteenth century; it did not fully displace the Julian calendar and gain universal acceptance until the early twentieth century. The Emacs calendar can display any month since January, year 1 of the current era, but the calendar displayed is the Gregorian, even for a date at which the Gregorian calendar did not exist.

While Emacs cannot display other calendars, it can convert dates to and from several other calendars.

29.8.1 Supported Calendar Systems

The ISO commercial calendar is used largely in Europe.

The Julian calendar, named after Julius Caesar, was the one used in Europe throughout medieval times, and in many countries up until the nineteenth century.

Astronomers use a simple counting of days elapsed since noon, Monday, January 1, 4713 B.C. on the Julian calendar. The number of days elapsed is called the *Julian day number* or the *Astronomical day number*.

The Hebrew calendar is the one used to determine the dates of Jewish holidays. Hebrew calendar dates begin and end at sunset.

The Islamic (Moslem) calendar is the one used to determine the dates of Moslem holidays. There is no universal agreement in the Islamic world about the calendar; Emacs uses a widely accepted version, but the precise dates of Islamic holidays often depend on proclamation by religious authorities, not on calculations. As a consequence, the actual dates of occurrence can vary slightly from the dates computed by Emacs. Islamic calendar dates begin and end at sunset.

The French Revolutionary calendar was created by the Jacobins after the 1789 revolution, to represent a more secular and nature-based view of the annual cycle, and to install a 10-day week in a rationalization measure similar to the metric system. The French government officially abandoned this calendar at the end of 1805.

The Maya of Central America used three separate, overlapping calendar systems, the *long count*, the *tzolkin*, and the *haab*. Emacs knows about all three of these calendars. Experts dispute the exact correlation between the Mayan calendar and our calendar; Emacs uses the Goodman-Martinez-Thompson correlation in its calculations.

29.8.2 Converting To Other Calendars

The following commands describe the selected date (the date at point) in various other calendar systems:

`p c`
: Display ISO commercial calendar equivalent for selected day (`calendar-print-iso-date`).

`p j`
: Display Julian date for selected day (`calendar-print-julian-date`).

`p a`
: Display astronomical (Julian) day number for selected day (`calendar-print-astro-day-number`).

`p h`
: Display Hebrew date for selected day (`calendar-print-hebrew-date`).

`p i`
: Display Islamic date for selected day (`calendar-print-islamic-date`).

`p f`
: Display French Revolutionary date for selected day (`calendar-print-french-date`).

`p m` Display Mayan date for selected day (`calendar-print-mayan-date`).

Put point on the desired date of the Gregorian calendar, then type the appropriate keys. The `p` is a mnemonic for "print" since Emacs "prints" the equivalent date in the echo area.

29.8.3 Converting From Other Calendars

You can move to dates that you specify on the Commercial, Julian, astronomical, Hebrew, Islamic, or French calendars:

`g c` Move to a date specified in the ISO commercial calendar (`calendar-goto-iso-date`).

`g j` Move to a date specified in the Julian calendar (`calendar-goto-julian-date`).

`g a` Move to a date specified in astronomical (Julian) day number (`calendar-goto-astro-day-number`).

`g h` Move to a date specified in the Hebrew calendar (`calendar-goto-hebrew-date`).

`g i` Move to a date specified in the Islamic calendar (`calendar-goto-islamic-date`).

`g f` Move to a date specified in the French Revolutionary calendar (`calendar-goto-french-date`).

These commands ask you for a date on the other calendar, move point to the Gregorian calendar date equivalent to that date, and display the other calendar's date in the echo area. Emacs uses strict completion (see Section 6.3 [Completion], page 37) whenever it asks you to type a month name, so you don't have to worry about the spelling of Hebrew, Islamic, or French names.

One common question concerning the Hebrew calendar is the computation of the anniversary of a date of death, called a "yahrzeit." The Emacs calendar includes a facility for such calculations. If you are in the calendar, the command `M-x list-yahrzeit-dates` asks you for a range of years and then displays a list of the yahrzeit dates for those years for the date given by point. If you are not in the calendar, this command first asks you for the date of death and the range of years, and then displays the list of yahrzeit dates.

29.8.4 Converting from the Mayan Calendar

Here are the commands to select dates based on the Mayan calendar:

`g m l` Move to a date specified by the long count calendar (`calendar-goto-mayan-long-count-date`).

`g m p t` Move to the previous occurrence of a place in the tzolkin calendar (`calendar-previous-tzolkin-date`).

`g m n t` Move to the next occurrence of a place in the tzolkin calendar (`calendar-next-tzolkin-date`).

`g m p h` Move to the previous occurrence of a place in the haab calendar (`calendar-previous-haab-date`).

`g m n h` Move to the next occurrence of a place in the haab calendar (`calendar-next-haab-date`).

`g m p c` Move to the previous occurrence of a place in the calendar round (`calendar-previous-calendar-round-date`).

`g m n c` Move to the next occurrence of a place in the calendar round (`calendar-next-calendar-round-date`).

To understand these commands, you need to understand the Mayan calendars. The *long count* is a counting of days with these units:

1 kin = 1 day   1 uinal = 20 kin   1 tun = 18 uinal
1 katun = 20 tun   1 baktun = 20 katun

Thus, the long count date 12.16.11.16.6 means 12 baktun, 16 katun, 11 tun, 16 uinal, and 6 kin. The Emacs calendar can handle Mayan long count dates as early as 7.17.18.13.1, but no earlier. When you use the `g m l` command, type the Mayan long count date with the baktun, katun, tun, uinal, and kin separated by periods.

The Mayan tzolkin calendar is a cycle of 260 days formed by a pair of independent cycles of 13 and 20 days. Since this cycle repeats endlessly, Emacs provides commands to move backward and forward to the previous or next point in the cycle. Type `g m p t` to go to the previous tzolkin date; Emacs asks you for a tzolkin date and moves point to the previous occurrence of that date. Similarly, type `g m n t` to go to the next occurrence of a tzolkin date.

The Mayan haab calendar is a cycle of 365 days arranged as 18 months of 20 days each, followed a 5-day monthless period. Like the tzolkin cycle, this cycle repeats endlessly, and there are commands to move backward and forward to the previous or next point in the cycle. Type `g m p h` to go to the previous haab date; Emacs asks you for a haab date and moves point to the previous occurrence of that date. Similarly, type `g m n h` to go to the next occurrence of a haab date.

The Maya also used the combination of the tzolkin date and the haab date. This combination is a cycle of about 52 years called a *calendar round*. If you type `g m p c`, Emacs asks you for both a haab and a tzolkin date and then moves point to the previous occurrence of that combination. Use `g m p c` to move point to the next occurrence of a combination. Emacs signals an error if the haab/tzolkin date you have typed is impossible.

Emacs uses strict completion (see Section 6.3 [Completion], page 37) whenever it asks you to type a Mayan name, so you don't have to worry about spelling.

29.9 The Diary

The Emacs diary keeps track of appointments or other events on a daily basis, in conjunction with the calendar. To use the diary feature, you must first create a *diary file* containing a list of events and their dates. Then Emacs can automatically pick out and display the events for today, for the immediate future, or for any specified date.

By default, Emacs uses '`~/diary`' as the diary file. This is the same file that the `calendar` utility uses. A sample '`~/diary`' file is:

```
12/22/1988 Twentieth wedding anniversary!!
&1/1. Happy New Year!
10/22 Ruth's birthday.
* 21, *: Payday
Tuesday--weekly meeting with grad students at 10am
         Supowit, Shen, Bitner, and Kapoor to attend.
1/13/89 Friday the thirteenth!!
&thu 4pm squash game with Lloyd.
mar 16 Dad's birthday
April 15, 1989 Income tax due.
&* 15 time cards due.
```

Although you probably will start by creating a diary manually, Emacs provides a number of commands to let you view, add, and change diary entries.

29.9.1 Commands Displaying Diary Entries

Once you have created a '`~/diary`' file, you can view it from within the calendar. You can also view today's events outside of Calendar mode.

`d` Display any diary entries for the selected date (`view-diary-entries`).

`s` Display entire diary file (`show-all-diary-entries`).

`m` Mark all visible dates that have diary entries (`mark-diary-entries`).

`u` Unmark calendar window (`calendar-unmark`).

`M-x print-diary-entries`
: Print hard copy of the diary display as it appears.

`M-x diary`
: Display any diary entries for today's date.

Displaying the diary entries with `d` shows in a separate window the diary entries for the selected date in the calendar. The mode line of the new window shows the date of the diary entries and any holidays that fall on that date.

If you specify a numeric argument with `d`, it shows all the diary entries for that many successive days. Thus, `2 d` displays all the entries for the selected date and for the following day.

To get a broader view of which days are mentioned in the diary, use the `m` command. This places a '`+`' after each date which has a diary entry. The command applies both to the currently visible months and to other months that subsequently become visible by scrolling. To turn marking off and erase the current marks, type `u`, which also turns off holiday marks (see Section 29.5 [Holidays], page 256).

To see the full diary file, rather than just some of the entries, use the `s` command.

Display of selected diary entries uses the selective display feature to hide entries that don't apply. This is the same feature that Outline mode uses to show part of an outline (see Section 21.8 [Outline Mode], page 152). The diary buffer as you see it is an illusion, so simply printing the buffer does not print what you see on your screen.

There is a special command to print hard copy of the diary buffer *as it appears*; this command is `M-x print-diary-entries`. It sends the data directly to the printer. You can customize it like `lpr-region` (see Section 30.5 [Hardcopy], page 283).

The command `M-x diary` displays the diary entries for the current date, independently of the calendar display, and optionally for the next few days as well; the variable `number-of-diary-entries` specifies how many days to include. See section "Calendar/Diary Options" in *The Emacs Lisp Reference Manual*.

If you put `(diary)` in your '`.emacs`' file, this automatically displays a window with the day's diary entries, when you enter Emacs. The mode line of the displayed window shows the date and any holidays that fall on that date.

29.9.2 The Diary File

Your *diary file* is a file that records events associated with particular dates. The name of the diary file is specified by the variable `diary-file`; '`~/diary`' is the default. You can use the same file for the `calendar` utility program, since its formats are a subset of the ones allowed by the Emacs diary facilities.

Each entry in the diary file describes one event and consists of one or more lines. An entry always begins with a date specification at the left margin. The rest of the entry is simply text to describe the event. If the entry has more than one line, then the lines after the first must begin with whitespace to indicate they continue a previous entry. Lines that do not begin with valid dates and do not continue a preceding entry are ignored.

You can inhibit the marking of certain diary entries in the calendar window; to do this, insert an ampersand (`&`) at the beginning of the entry, before the date. This has no effect on display of the entry in the diary window; it affects only marks on dates in the calendar window. Nonmarking entries are especially useful for generic entries that would otherwise mark many different dates.

If the first line of a diary entry consists only of the date or day name with no following blanks or punctuation, then the diary window display doesn't include that line; only the continuation lines appear. For example:

```
02/11/1989
      Bill B. visits Princeton today
      2pm Cognitive Studies Committee meeting
      2:30-5:30 Liz at Lawrenceville
      4:00pm Dentist appt
      7:30pm Dinner at George's
      8:00-10:00pm concert
```

appears in the diary window without the date line at the beginning. This style of entry looks neater when you display just a single day's entries, but can cause confusion if you ask for more than one day's entries.

You can edit the diary entries as they appear in the window, but it is important to remember that the buffer displayed contains the *entire* diary file, with portions of it concealed from view. This means, for instance, that the `C-f` (`forward-char`) command can put point at what appears to be the end of the line, but what is in reality the middle of some concealed line.

Be careful when editing the diary entries! Inserting additional lines or adding/deleting characters in the middle of a visible line cannot cause problems, but editing at the end of a line may not do what you expect. Deleting a line may delete other invisible entries that follow it. Before editing the diary, it is best to display the entire file with `s` (`show-all-diary-entries`).

29.9.3 Date Formats

Here are some sample diary entries, illustrating different ways of formatting a date. The examples all show dates in American order (month, day, year), but Calendar mode supports European order (day, month, year) as an option.

```
4/20/93  Switch-over to new tabulation system
apr. 25  Start tabulating annual results
4/30  Results for April are due
*/25  Monthly cycle finishes
Friday  Don't leave without backing up files
```

The first entry appears only once, on April 20, 1993. The second and third appear every year on the specified dates, and the fourth uses a wildcard (asterisk) for the month, so it appears on the 25th of every month. The final entry appears every week on Friday.

You can use just numbers to express a date, as in '*month*/*day*' or '*month*/*day*/*year*'. This must be followed by a nondigit. In the date itself, *month* and *day* are numbers of one or two digits. *year* is a number and may be abbreviated to the last two digits; that is, you can use '`11/12/1989`' or '`11/12/89`'.

Dates can also have the form '*monthname day*' or '*monthname day*, *year*', where the month's name can be spelled in full or abbreviated to three characters (with or without a period). Case is not significant.

A date may be *generic*, or partially unspecified. Then the entry applies to all dates that match the specification. If the date does not contain a year, it is generic and applies to any year. Alternatively, *month*, *day*, or *year* can be a '`*`'; this matches any month, day, or year, respectively. Thus, a diary entry '`3/*/*`' matches any day in March of any year; so does '`march *`'.

If you prefer the European style of writing dates—in which the day comes before the month—type `M-x european-calendar` while in the calendar, or set the variable `european-calendar-style` to `t` *before* using any calendar or diary command. This mode interprets all dates in the diary in the European manner, and also uses European style for displaying diary dates. (Note that there is no comma after the *monthname* in the European style.) To go back to the (default) American style of writing dates, type `M-x american-calendar`.

You can use the name of a day of the week as a generic date which applies to any date falling on that day of the week. You can abbreviate the day of the week to three letters (with or without a period) or spell it in full; case is not significant.

29.9.4 Commands to Add to the Diary

While in the calendar, there are several commands to create diary entries:

`i d` Add a diary entry for the selected date (`insert-diary-entry`).

`i w` Add a diary entry for the selected day of the week (`insert-weekly-diary-entry`).

`i m` Add a diary entry for the selected day of the month (`insert-monthly-diary-entry`).

`i y` Add a diary entry for the selected day of the year (`insert-yearly-diary-entry`).

You can make a diary entry for a specific date by selecting that date in the calendar window and typing the `i d` command. This command displays the end of your diary file in another window and inserts the date; you can then type the rest of the diary entry.

If you want to make a diary entry that applies to a specific day of the week, select that day of the week (any occurrence will do) and type `i w`. This inserts the day-of-week as a generic date; you can then type the rest of the diary entry.

You can make a monthly diary entry in the same fashion. Select the day of the month, use the `i m` command, and type rest of the entry. Similarly, you can insert a yearly diary entry with the `i y` command.

All of the above commands make marking diary entries. If you want the diary entry to be nonmarking, give a prefix argument to the command. For example, `C-u i w` makes a nonmarking, weekly diary entry.

When you modify the diary file, be sure to save the file before exiting Emacs.

29.9.5 Special Diary Entries

In addition to entries based on calendar dates, your diary file can contain *special entries* for regular events such as anniversaries. These entries are based on Lisp expressions (sexps) that Emacs evaluates as it scans the diary file. Instead of a date, a special entry contains '`%%`' followed by a Lisp expression which must begin and end with parentheses. The Lisp expression determines which dates the entry applies to.

Calendar mode provides commands to insert certain commonly used special entries:

`i a` Add an anniversary diary entry for the selected date (`insert-anniversary-diary-entry`).

`i b` Add a block diary entry for the current region (`insert-block-diary-entry`).

`i c` Add a cyclic diary entry starting at the date (`insert-cyclic-diary-entry`).

If you want to make a diary entry that applies to the anniversary of a specific date, move point to that date and use the `i a` command. This displays the end of your diary file in another window and inserts the anniversary description; you can then type the rest of the diary entry. The entry looks like this:

```
%%(diary-anniversary 10 31 1948) Arthur's birthday
```

This entry applies to October 31 in any year after 1948; '`10 31 1948`' specifies the date. (If you are using the European calendar style, the month and day are interchanged.) The reason this expression requires a beginning year is that advanced diary functions can use it to calculate the number of elapsed years.

You can make a diary entry entry for a block of dates by setting the mark at the date at one end of the block, moving point to the date at the other end of the block, and using the `i b` command. This command causes the end of your diary file to be displayed in another window and the block description to be inserted; you can then type the diary entry. Here is a block diary entry that applies to all dates from June 24, 1990 through July 10, 1990:

```
%%(diary-block 6 24 1990 7 10 1990) Vacation
```

The '`6 24 1990`' indicates the starting date and the '`7 10 1990`' indicates the stopping date. (Again, if you are using the European calendar style, the month and day are interchanged.)

Cyclic diary entries repeat after a fixed interval of days. To create one, select the starting date and use the `i c` command. The command prompts for the length of interval, then inserts the entry. It looks like this:

```
%%(diary-cyclic 50 3 1 1990) Renew medication
```

which applies to March 1, 1990 and every 50th day following; '`3 1 1990`' specifies the starting date. (If you are using the European calendar style, the month and day are interchanged.)

All three of the these commands make marking diary entries. If you want the diary entry to be nonmarking, give a numeric argument to the command. For example, `C-u i a` makes a nonmarking anniversary diary entry.

Marking sexp diary entries in the calendar is *extremely* time-consuming, since every date visible in the calendar window must be individually checked. So it's a good idea to make sexp diary entries nonmarking (with '`&`') when possible.

Another sophisticated kind of sexp entry, a *floating* diary entry, specifies a regularly-occurring event by offsets specified in days, weeks, and months. It is comparable to a crontab entry interpreted by the `cron` utility. Here is a nonmarking, floating diary entry that applies to the last Thursday in November:

```
&%%(diary-float 11 4 -1) American Thanksgiving
```

The 11 specifies November (the eleventh month), the 4 specifies Thursday (the fourth day of the week, where Sunday is numbered zero), and the −1 specifies "last" (1 would mean "first", 2 would mean "second", −2 would mean "second-to-last", and so on). The month can be a single month or a list of months. Thus you could change the 11 above to `'(1 2 3)'` and have the entry apply to the last Thursday of January, February, and March. If the month is `t`, the entry applies to all months of the year.

Most generally, special diary entries can perform arbitrary computations to determine when they apply. See section "Sexp Diary Entries" in *The Emacs Lisp Reference Manual*.

29.10 Appointments

If you have a diary entry for an appointment, and that diary entry begins with a recognizable time of day, Emacs can warn you, several minutes beforehand, that that appointment is pending. Emacs alerts you to the appointment by displaying a message in the mode line.

To enable appointment notification, you must enable the time display feature of Emacs, `M-x display-time` (see Chapter 2 [Mode Line], page 19). You must also add the function `appt-make-list` to the `diary-display-hook`, like this:

```
(add-hook 'diary-display-hook 'appt-make-list)
```

With these preparations done, when you display the diary (either with the `d` command in the calendar window or with the `M-x diary` command), it sets up an appointment list of all the diary entries found with recognizable times of day, and reminds you just before each of them.

For example, if you the diary file contains these lines:

```
Monday
  9:30am Coffee break
 12:00pm Lunch
```

Then on Mondays, after you have displayed the diary, you will be reminded at 9:20am about your coffee break and at 11:50am about lunch.

Diary entries can have the time in the conventional American style, or in "military" style. You need not be consistent; your diary file can have a mixture of the two styles.

Emacs updates the appointments list automatically just after midnight. This also displays the next days' diary entries in the diary buffer, unless you set `appt-display-diary` to `nil`.

You can also use the appointment notification facility like an alarm clock. The command `M-x appt-add` adds entries to the appointment list without affecting your diary file. You delete entries from the appointment list with `M-x appt-delete`.

You can turn off the appointment notification feature at any time by setting `appt-issue-message` to `nil`.

29.11 Daylight Savings Time

Emacs understands the difference between standard time and daylight savings time—the times given for sunrise, sunset, solstices, equinoxes, and the phases of the moon take that into account. The rules for daylight savings time vary from place to place and have also varied historically from year to year. To do the job properly, Emacs needs to know which rules to use.

Some operating systems keep track of the rules that apply to the place where you are; on these systems, Emacs gets the information it needs from the system automatically. If some or all of this information is missing, Emacs fills in the gaps with the rules currently used in Cambridge, Massachusetts. If the resulting rules are not what you want, you can tell Emacs the rules to use by setting certain variables.

These values should be Lisp expressions that refer to the variable `year`, and evaluate to the Gregorian date on which daylight savings time starts or (respectively) ends, in the form of a list `(`*month day year*`)`. The values should be `nil` if your area does not use daylight savings time.

Emacs uses these expressions to determine the starting date of daylight savings time for the holiday list and for correcting times of day in the solar and lunar calculations.

The values for Cambridge, Massachusetts are as follows:

```
(calendar-nth-named-day 1 0 4 year)
(calendar-nth-named-day -1 0 10 year)
```

i.e. the first 0th day (Sunday) of the fourth month (April) in the year specified by `year`, and the last Sunday of the tenth month (October) of that year. If daylight savings time were changed to start on October 1, you would set `calendar-daylight-savings-starts` to this:

```
(list 10 1 year)
```

If there is no daylight savings time at your location, or if you want all times in standard time, set `calendar-daylight-savings-starts` and `calendar-daylight-savings-ends` to `nil`.

This variable specifies the difference between daylight savings time and standard time, measured in minutes. The value for Cambridge is 60.

These variables specify is the number of minutes after midnight local time when the transition to and from daylight savings time should occur. For Cambridge, both variables' values are 120.

30 Miscellaneous Commands

This chapter contains several brief topics that do not fit anywhere else.

30.1 GNUS

GNUS is an Emacs subsystem for reading and responding to netnews. You can use GNUS to browse through news groups, look at summaries of articles in specific group, and read articles of interest. You can respond to authors or write replies to all the readers of a news group.

This section introduces GNUS and describes several basic features. Full documentation will appear elsewhere.

To start GNUS, type `M-x gnus RET`.

30.1.1 GNUS's Three Buffers

GNUS creates and uses three Emacs buffers, each with its own particular purpose and its own major mode.

The *Newsgroup buffer* contains a list of newsgroups. This is the first buffer that GNUS displays when it starts up. Normally the list contains only the newsgroups to which you subscribe (which are listed in your '`.newsrc`' file) and which contain unread articles. Use this buffer to select a specific newsgroup.

The *Summary buffer* lists the articles in a single newsgroup, including their subjects, their numbers, and who posted them. GNUS creates a Summary buffer for a newsgroup when you select the group in the Newsgroup buffer. Use this buffer to select an article, and to move around in an article.

The *Article buffer* displays the text of an article. You rarely need to select this buffer because you can read the text while keeping the Summary buffer selected.

30.1.2 When GNUS Starts Up

At startup, GNUS reads your '`.newsrc`' news initialization file and attempts to communicate with the local news server, which is a repository of news articles. The news server need not be the same computer you are logged in on.

If you start GNUS and connect to the server, but do not see any newsgroups listed in the Newsgroup buffer, type `L` to get a listing of all the newsgroups. Then type `u` to unsubscribe from particular newsgroups. (Move the cursor using `n` and `p` or the usual Emacs commands.)

When you quit GNUS with `q`, it automatically records in your '`.newsrc`' initialization file the subscribed or unsubscribed status of all newsgroups, except for groups you have "killed". (You do not need to edit this file yourself, but you may.) When new newsgroups come into existence, GNUS adds them automatically.

30.1.3 Summary of GNUS Commands

Reading news is a two step process:

1. Choose a newsgroup in the Newsgroup buffer.
2. Select articles from the Summary buffer. Each article selected is displayed in the Article buffer in a large window, below the Summary buffer in its small window.

Each buffer has commands particular to it, but commands that do the same things have similar keybindings. Here are commands for the Newsgroup and Summary buffers:

`z`
: In the Newsgroup buffer, suspend GNUS. You can return to GNUS later by selecting the Newsgroup buffer and typing `g` to get newly arrived articles.

`q`
: In the Newsgroup buffer, update your '`.newsrc`' initialization file and quit GNUS.

  In the Summary buffer, exit the current newsgroup and return to the Newsgroup buffer. Thus, typing `q` twice quits GNUS.

`L`
: In the Newsgroup buffer, list all the newsgroups available on your news server. This may be a long list!

`l`
: In the Newsgroup buffer, list only the newsgroups to which you subscribe and which contain unread articles.

`u`
: In the Newsgroup buffer, unsubscribe from (or subscribe to) the newsgroup listed in the line that point is on. When you quit GNUS by typing `q`, GNUS lists your subscribed-to newsgroups in your '`.newsrc`' file. The next time you start GNUS, you see only the newsgroups listed in your '`.newsrc`' file.

`C-k`
: In the Newsgroup buffer, "kill" the current line's newsgroup—don't show it in the Newsgroup buffer from now on. This affects future GNUS sessions as well as the present session.

  When you quit GNUS by typing `q`, GNUS writes information in the file '`.newsrc`' describing all newsgroups except those you have "killed."

`SPC`
: In the Newsgroup buffer, select the group on the line under the cursor and display the first unread article in that group.

In the Summary buffer,

- Select the article on the line under the cursor if none is selected.
- Scroll the text of the selected article (if there is one).
- Select the next unread article if at the end of the current article.

Thus, you can move through all the articles by repeatedly typing `SPC`.

`DEL` In the Newsgroup Buffer, move point to the previous newsgroup containing unread articles.

In the Summary buffer, scroll the text of the article backwards.

`n` Move point to the next unread newsgroup, or select the next unread article.

`p` Move point to the previous unread newsgroup, or select the previous unread article.

`C-n`
`C-p` Move point to the next or previous item, even if it is marked as read. This does not select the article or newsgroup on that line.

`s` In the Summary buffer, do an incremental search of the current text in the Article buffer, just as if you switched to the Article buffer and typed `C-s`.

`M-s` *regexp* `RET`
In the Summary buffer, search forward for articles containing a match for *regexp*.

`C-c C-s C-n`
`C-c C-s C-s`
`C-c C-s C-d`
`C-c C-s C-a`
In the Summary buffer, sort the list of articles by number, subject, date, or author.

`C-M-n`
`C-M-p` In the Summary buffer, read the next or previous article with the same subject as the current article.

30.2 Sorting Text

Emacs provides several commands for sorting text in the buffer. All operate on the contents of the region (the text between point and the mark).

They divide the text of the region into many *sort records*, identify a *sort key* for each record, and then reorder the records into the order determined by the sort keys. The records are ordered so that their keys are in alphabetical order, or, for numeric sorting, in numeric order. In alphabetic sorting, all upper case letters 'A' through 'Z' come before lower case 'a', in accord with the ASCII character sequence.

The various sort commands differ in how they divide the text into sort records and in which part of each record is used as the sort key. Most of the commands make each line a separate sort record, but some commands use paragraphs or pages as sort records. Most of the sort commands use each entire sort record as its own sort key, but some use only a portion of the record as the sort key.

`M-x sort-lines`
: Divide the region into lines, and sort by comparing the entire text of a line. A prefix argument means sort into descending order.

`M-x sort-paragraphs`
: Divide the region into paragraphs, and sort by comparing the entire text of a paragraph (except for leading blank lines). A prefix argument means sort into descending order.

`M-x sort-pages`
: Divide the region into pages, and sort by comparing the entire text of a page (except for leading blank lines). A prefix argument means sort into descending order.

`M-x sort-fields`
: Divide the region into lines, and sort by comparing the contents of one field in each line. Fields are defined as separated by whitespace, so the first run of consecutive non-whitespace characters in a line constitutes field 1, the second such run constitutes field 2, etc.

  Specify which field to sort by with a numeric argument: 1 to sort by field 1, etc. A negative argument means sort into descending order. Thus, minus 2 means sort by field 2 in reverse-alphabetical order. 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 a number for each line, and the numbers are compared. '`10`' comes before '`2`' when considered as text, but after it when considered as a number.

`M-x sort-columns`
: Like `M-x sort-fields` except that the text within each line used for comparison comes from a fixed range of columns. See below for an explanation.

For example, if the buffer contains 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.
```

then 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 to sort, this command uses an unusual definition of 'region': all of the line point is in is considered part of the region, and so is all of the line the mark is in.

For example, to sort a table by information found in columns 10 to 15, you could put the mark on column 10 in the first line of the table, and point on column 15 in the last line of the table, and then 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 10.4 [Rectangles], page 63.

Many of the sort commands ignore case differences when comparing, if `sort-fold-case` is non-`nil`.

30.3 Running Shell Commands from Emacs

Emacs has commands for passing single command lines to inferior shell processes; it can also run a shell interactively with input and output to an Emacs buffer '*shell*'.

`M-!` Run a specified shell command line and display the output (`shell-command`).

`M-|` Run a specified shell command line with region contents as input; optionally replace the region with the output (`shell-command-on-region`).

`M-x shell` Run a subshell with input and output through an Emacs buffer. You can then give commands interactively.

30.3.1 Single Shell Commands

`M-!` (`shell-command`) reads a line of text using the minibuffer executes it as a shell command in a subshell made just for this command. Standard input for the command comes from the null device. If the shell command produces any output, the output goes into an Emacs buffer named '*Shell Command Output*', which is displayed in another window but not selected. A numeric argument, as in `M-1 M-!`, directs this command to insert any output into the current buffer. In that case, point is left before the output and the mark is set after the output.

If the shell command line ends in '&', it runs asynchronously.

`M-|` (`shell-command-on-region`) is like `M-!` but passes the contents of the region as input to the shell command, instead of no input. If a numeric argument is used, meaning insert output in the current buffer, then the old region is deleted first and the output replaces it as the contents of the region.

Both `M-!` and `M-|` use `shell-file-name` to specify the shell to use. This variable is initialized based on your `SHELL` environment variable when Emacs is started. If the file name does not specify a directory, the directories in the list `exec-path` are searched; this list is initialized based on the environment variable `PATH` when Emacs is started. Your '.emacs' file can override either or both of these default initializations.

With `M-!` and `M-|`, Emacs has to wait until the shell command completes. To stop waiting, type `C-g` to quit; that also kills the shell command.

30.3.2 Interactive Inferior Shell

To run a subshell interactively, putting its typescript in an Emacs buffer, use `M-x shell`. This creates (or reuses) a buffer named '*shell*' and runs a subshell with input coming from and output going to that buffer. That is to say, any "terminal output" from the subshell goes into the buffer, advancing point, and any "terminal input" for the subshell comes from text in the buffer. To give input to the subshell, go to the end of the buffer and type the input, terminated by `RET`.

Emacs does not wait for the subshell to do anything. You can switch windows or buffers and edit them while the shell is waiting, or while it is running a command. Output from the subshell waits until Emacs has time to process it; this happens whenever Emacs is waiting for keyboard input or for time to elapse.

To make multiple subshells, rename the buffer '`*shell*`' to something different using `M-x rename-uniquely`. Then type `M-x shell` again to create a new buffer '`*shell*`' with its own subshell. If you rename this buffer as well, you can create a third one, and so on. All the subshells run independently and in parallel.

The file name used to load the subshell is the value of the variable `explicit-shell-file-name`, if that is non-`nil`. Otherwise, the environment variable `ESHELL` is used, or the environment variable `SHELL` if there is no `ESHELL`. If the file name specified is relative, the directories in the list `exec-path` are searched (see Section 30.3.1 [Single Shell], page 276).

As soon as the subshell is started, it is sent as input the contents of the file '`~/.emacs_`*shellname*', if that file exists, where *shellname* is the name of the file that the shell was loaded from. For example, if you use `bash`, the file sent to it is '`~/.emacs_bash`'.

`cd`, `pushd` and `popd` commands given to the inferior shell are watched by Emacs so it can keep the '`*shell*`' buffer's default directory the same as the shell's working directory. These commands are recognized syntactically by examining lines of input that are sent. If you use aliases for these commands, you can tell Emacs to recognize them also. For example, if the value of the variable `shell-pushd-regexp` matches the beginning of a shell command line, that line is regarded as a `pushd` command. Change this variable when you add aliases for '`pushd`'. Likewise, `shell-popd-regexp` and `shell-cd-regexp` are used to recognize commands with the meaning of '`popd`' and '`cd`'. These commands are recognized only at the beginning of a shell command line.

If Emacs gets an error while trying to handle what it believes is a '`cd`', '`pushd`' or '`popd`' command, it runs the hook `shell-set-directory-error-hook` (see Section 31.2.3 [Hooks], page 294).

If Emacs does not properly track changes in the current directory of the subshell, use the command `M-x dirs` to ask the shell what its current directory is. This command works for shells that support the most common command syntax; it may not work for unusual shells.

30.3.3 Shell Mode

The shell buffer uses Shell mode, which defines several special keys attached to the `C-c` prefix. They are chosen to resemble the usual editing and job control characters present in shells that are not under Emacs, except that you must type `C-c` first. Here is a complete list of the special key bindings of Shell mode:

`RET` At end of buffer send line as input; otherwise, copy current line to end of buffer and send it (`comint-send-input`). When a line is copied, any text at the beginning of the line that matches the variable `shell-prompt-pattern` is left out; this variable's value should be a regexp string that matches the prompts that your shell uses.

`TAB` Complete the command name or file name before point in the shell buffer (`comint-dynamic-complete`). `TAB` also completes history references; see Section 30.3.4.3 [History References], page 281.

`M-?` Display temporarily a list of the possible completions of the file name before point in the shell buffer (`comint-dynamic-list-filename-completions`).

`C-a` Move to the beginning of the line, but after the prompt if any (`comint-bol`).

`C-d` Either delete a character or send EOF (`comint-delchar-or-maybe-eof`). Typed at the end of the shell buffer, `C-d` sends EOF to the subshell. Typed at any other position in the buffer, `C-d` deletes a character as usual.

`C-c C-u` Kill all text pending at end of buffer to be sent as input (`comint-kill-input`).

`C-c C-w` Kill a word before point (`backward-kill-word`).

`C-c C-c` Interrupt the shell or its current subjob if any (`comint-interrupt-subjob`).

`C-c C-z` Stop the shell or its current subjob if any (`comint-stop-subjob`).

`C-c C-\` Send quit signal to the shell or its current subjob if any (`comint-quit-subjob`).

`C-c C-o` Kill the last batch of output from a shell command (`comint-kill-output`). This is useful if a shell command spews out lots of output that just gets in the way.

`C-c C-r` Scroll to display the beginning of the last batch of output at the top of the window; also move the cursor there (`comint-show-output`).

`C-c C-e` Scroll to put the end of the buffer at the bottom of the window (`comint-show-maximum-output`).

`C-c C-f` Move forward across one shell command, but not beyond the current line (`shell-forward-command`). The variable `shell-command-regexp` specifies how to recognize the end of a command.

`C-c C-b` Move backward across one shell command, but not beyond the current line (`shell-backward-command`).

`M-x dirs` Ask the shell what its current directory is, so that Emacs can agree with the shell.

`M-x send-invisible RET` *text* `RET`
: Send *text* as input to the shell, after reading it without echoing. This is useful when a shell command runs a program that asks for a password.

`M-x comint-continue-subjob`
: Continue the shell process. This is useful if you accidentally suspend the shell process.[1]

Shell mode also customizes the paragraph commands so that only shell promps start new paragraphs. Thus, a paragraph consists of an input command plus the output that follows it in the buffer.

30.3.4 Shell Command History

Shell buffers support three ways of repeating earlier commands. You can use the same keys used in the minibuffer; these work much as they do in the minibuffer, inserting text from prior commands while point remains always at the end of the buffer. You can move through the buffer to previous inputs in their original place, then resubmit them or copy them to the end. Or you can use a '`!`'-style history reference.

30.3.4.1 Shell History Ring

`M-p` Fetch the next earlier old shell command.

[1] 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.

`M-n` Fetch the next later old shell command.

`M-r` *regexp* `RET`
`M-s` *regexp* `RET`
Search backwards or forwards for old shell commands that match *regexp*.

Shell buffers provide a history of previously entered shell commands. To reuse shell commands from the history, use the editing commands `M-p`, `M-n`, `M-r` and `M-s`. These work just like the minibuffer history commands except that they operate on the text at the end of the shell buffer, where you would normally insert text to send to the shell.

`M-p` fetches an earlier shell command to the end of the shell buffer. Successive use of `M-p` fetches successively earlier shell commands, each replacing any text that was already present as potential shell input. `M-n` does likewise except that it finds successively more recent shell commands from the buffer.

The history search commands `M-r` and `M-s` read a regular expression and search through the history for a matching command. Aside from the choice of which command to fetch, they work just like `M-p` and `M-r`. If you enter an empty regexp, these commands reuse the same regexp used last time.

When you find the previous input you want, you can resubmit it by typing `RET`, or you can edit it first and then resubmit it if you wish.

These commands get the text of previous shell commands from a special history list, not from the shell buffer itself. Thus, editing the shell buffer, or even killing large parts of it, does not affect the history that these commands access.

30.3.4.2 Shell History Copying

`C-c C-p` Move point to the previous prompt (`comint-previous-prompt`).

`C-c C-n` Move point to the following prompt (`comint-next-prompt`).

`C-c RET` Copy the input command which point is in, inserting the copy at the end of the buffer (`comint-copy-old-input`). This is useful if you move point back to a previous command. After you copy the command, you can submit the copy as input with `RET`. If you wish, you can edit the copy before resubmitting it.

Moving to a previous input and then copying it with `C-c RET` produces the same results—the same buffer contents—that you would get by using `M-p` enough times to fetch that previous input from the history list. However, `C-c RET` copies the text from the buffer, which can be different from what is in the history list (if you edit the input in the buffer after it has been sent).

30.3.4.3 Shell History References

Various shells including csh and bash support *history references* that begin with '!' and '^'. Shell mode now understands this syntax. If you insert a history reference and type TAB, this searches the input history for a matching command, performs substitution if necessary, and places the result in the buffer in place of the history reference. For example, you can fetch the most recent command beginning with 'mv' with ! m v TAB. You could then resubmit this command to the shell by typing RET as usual.

30.3.5 Shell Mode Options

If the variable comint-scroll-to-bottom-on-input is non-nil, insertion and yank commands scroll the selected window the bottom before inserting.

If comint-scroll-show-maximum-output is non-nil (which is the default), then scrolling due to arrival of output tries to place the last line of text at the bottom line of the window, so as to show as much useful text as possible. (This mimics the scrolling behavior of many terminals.)

By setting comint-scroll-to-bottom-on-output, you can opt for having point jump to the end of the buffer whenever output arrives—no matter where in the buffer point was before. If the value is this, point jumps in the selected window. If the value is all, point jumps in each window that shows the comint buffer. If the value is other, point jumps in all nonselected windows that show the current buffer. The default value is nil, which means point does not jump to the end.

The variable comint-input-ignoredups controls whether successive identical inputs are stored in the input history. A non-nil value means to omit an input that is the same as the previous input. The default is nil, which means to store each input even if it is equal to the previous input.

Three variables customize filename completion comint-completion-addsuffix controls whether completion inserts a space or a slash to indicate a fully completed file or directory name (non-nil means do insert a space or slash). comint-completion-recexact, if non-nil, directs TAB to choose the shortest possible completion if the usual Emacs completion algorithm cannot add even a single character. comint-completion-autolist, if non-nil, says to list all the possible completions whenever completion is not exact.

Shell mode can optionally expand history references in the buffer when you send them to the shell. To request this, set the variable comint-input-autoexpand to input.

You can make SPC perform history expansion by binding SPC to the command comint-magic-space.

The command `comint-dynamic-complete-variable` does variable name completion using the environment variables as set within Emacs. The variables controlling filename completion apply to variable name completion too. This command is normally available through the menu bar.

Command completion normally considers only executable files. If you set `shell-command-execonly` to `nil`, it considers nonexecutable files as well.

In a new Shell mode buffer, the input history is initialized from the file name specified in the variable `shell-input-ring-file-name`. By default, this is `"~/.history"`.

You can now configure the behavior of '`pushd`'. Variables control whether '`pushd`' behaves like '`cd`' if no argument is given (`shell-pushd-tohome`), pop rather than rotate with a numeric argument (`shell-pushd-dextract`), and only add directories to the directory stack if they are not already on it (`shell-pushd-dunique`). The values you choose should match the underlying shell, of course.

30.4 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*. Cancelling the narrowing, and making 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*.

`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`).

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. In addition to sometimes making it easier to concentrate on a single subroutine or paragraph by eliminating clutter, narrowing can be used to restrict the range of operation of a replace command or repeating keyboard macro. The word '`Narrow`' appears in the mode line whenever narrowing is in effect.

The primary narrowing command is `C-x n 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 invisible. Point and mark do not change.

Alternatively, use `C-x n p` (`narrow-to-page`) to narrow down to the current page. See Section 21.4 [Pages], page 146, for the definition of a page.

The way to undo 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 4.8 [Position Info], page 30.

Because narrowing can easily confuse users who do not understand it, `narrow-to-region` is normally a disabled command. Attempting to use this command asks for confirmation and gives you the option of enabling it; once you enable the command, confirmation will no longer be required for it. See Section 31.4.9 [Disabling], page 310.

30.5 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 15.10 [Misc File Ops], page 116) and the diary (see Section 29.9.1 [Diary Commands], page 263).

`M-x print-buffer`
: Print hardcopy of current buffer using Unix command '`print`' ('`lpr -p`'). This makes page headings containing the file name and page number.

`M-x lpr-buffer`
: Print hardcopy of current buffer using Unix command '`lpr`'. This makes no page headings.

`M-x print-region`
: Like `print-buffer` but prints only the current region.

`M-x lpr-region`
: Like `lpr-buffer` but prints only the current region.

All the hardcopy commands pass extra switches to the `lpr` program based on the value of the variable `lpr-switches`. Its value should be a list of strings, each string an option starting with '`-`'. For example, to use a printer named '`nearme`', set `lpr-switches` like this:

```
(setq lpr-switches '("-Pnearme"))
```

30.6 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:

`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 (`tc-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 contains just one column and you want to add another column.

`C-x 6 s` Split the current buffer, which contains two-column text, into two buffers, and display them side by side (`tc-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.

`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 (`tc-associate-buffer`).

`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 `C-x 6 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, `C-x 6 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 which "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.)

It's not a good idea to use ordinary scrolling commands during two-column editing, because that separates the two parts of each split line. Instead, use these special scroll commands:

`C-x 6 SPC` Scroll both buffers up, in lockstep (`tc-scroll-up`).

`C-x 6 DEL` Scroll both buffers down, in lockstep (`tc-scroll-down`).

`C-x 6 C-l` Recenter both buffers, in lockstep (`tc-recenter`).

When you have edited both buffers as you wish, merge them with `C-x 6 1` (`tc-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 `C-x 6 s`.

Use `C-x 6 d` to disassociate the two buffers, leaving each as it stands (`tc-dissociate`). If the other buffer, the one not current when you type `C-x 6 d`, is empty, `C-x 6 d` kills it.

30.7 Editing Binary Files

There is a special major mode for editing binary files: Hexl mode. To use it, use `M-x hexl-find-file` instead of `C-x C-f` to visit the file. This command converts the file's contents to hexadecimal and lets you edit the translation. When you save the file, it is converted automatically back to binary.

You can also use `M-x hexl-mode` to translate an existing buffer into hex. This is useful if you visit a file normally and then discover it is a binary file.

Ordinary text characters overwrite in Hexl mode. This is to reduce the risk of accidentally spoiling the alignment of data in the file. There are special commands for insertion. Here is a list of the commands of Hexl mode:

`C-M-d`
: Insert a byte with a code typed in decimal.

`C-M-o`
: Insert a byte with a code typed in octal.

`C-M-x`
: Insert a byte with a code typed in hex.

`C-x [`
: Move to the beginning of a 1k-byte "page".

`C-x ]`
: Move to the end of a 1k-byte "page".

`M-g`
: Move to an address specified in hex.

`M-j`
: Move to an address specified in decimal.

`C-c C-c`
: Leave Hexl mode, going back to the major mode this buffer had before you invoked `hexl-mode`.

30.8 Using Emacs as a Server

Various programs such as `mail` can invoke your choice of editor to edit a particular piece of text, such as a message that you are sending. By convention, these programs use the environment variable `EDITOR` to specify which editor to run. If you set `EDITOR` to '`emacs`', they invoke Emacs—but in an inconvenient fashion, by starting a new, separate Emacs process. This is inconvenient because it takes time and because the new Emacs process doesn't share the buffers in the existing Emacs process.

You can arrange to use your existing Emacs process as the editor for programs like `mail` by using the Emacs client and Emacs server programs. Here is how.

First, the preparation. Within Emacs, call the function `server-start`. (Your '`.emacs`' file can do this automatically if you add the expression `(server-start)` to it.) Then, outside Emacs, set the `EDITOR` environment variable to '`emacsclient`'.

Then, whenever any program invokes your specified `EDITOR` program, the effect is to send a message to your principal Emacs telling it to visit a file. (That's what the program `emacsclient` does.) Emacs obeys silently; it does not immediately switch to the new file's buffer. When you want to do that, type `C-x #` (`server-edit`).

When you've finished editing that buffer, type `C-x #` again. This saves the file and sends a message back to the `emacsclient` program telling it to exit. The programs that use `EDITOR` wait for the "editor" (actually, `emacsclient`) to exit. `C-x #` also checks to see if any other files are pending for you to edit, and selects the next one.

You can switch to a server buffer manually if you wish; you don't have to arrive at it with `C-x #`. But `C-x #` is the only way to say that you are "finished" with one.

If you set the variable `server-window` to a window or a frame, `C-x #` displays the server buffer in that window or in that frame.

While `mail` or another application is waiting for `emacsclient` to finish, `emacsclient` does not read terminal input. So the terminal that `mail` was using is effectively blocked for the duration. In order to edit with your principal Emacs, you need to be able to use it without using that terminal. There are two ways to do this:

- Using a window system, run `mail` and the principal Emacs in two separate windows. While `mail` is waiting for `emacsclient`, the window where it was running is blocked, but you can use Emacs by switching windows.
- Use Shell mode in Emacs to run the other program such as `mail`; then, `emacsclient` blocks only the subshell under Emacs; you can still use Emacs to edit the file.

Some programs write temporary files for you to edit. After you edit the temporary file, the program reads it back and deletes it. If the Emacs server is later asked to edit the same file name, it should assume this has nothing to do with the previous occasion for that file name. The server accomplishes this by killing the temporary file's buffer when you finish with the file. Use the variable `server-temp-file-regexp` to specify which files are temporary in this sense; its value should be a regular expression that matches file names that are temporary.

30.9 Recursive Editing Levels

A *recursive edit* is a situation in which you are using Emacs commands to perform arbitrary editing while in the middle of another Emacs command. For example, when you type `C-r` inside of a `query-replace`, you enter a recursive edit in which you can change the current buffer. On exiting from the recursive edit, you go back to the `query-replace`.

Exiting the recursive edit means returning to the unfinished command, which continues execution. To exit, type `C-M-c` (`exit-recursive-edit`).

You can also *abort* the recursive edit. This is like exiting, but also quits the unfinished command immediately. Use the command `C-]` (`abort-recursive-edit`) for this. See Section 32.1 [Quitting], page 317.

The mode line shows you when you are in a recursive edit by displaying square brackets around the parentheses that always surround the major and minor mode names. Every window's mode line shows this, in the same way, since being in a recursive edit is true of Emacs as a whole rather than any particular window or buffer.

It is possible to be in recursive edits within recursive edits. For example, after typing `C-r` in a `query-replace`, you may type a command that enters the debugger. This begins a recursive editing level for the debugger, within the recursive editing level for `C-r`. Mode lines display a pair of square brackets for each recursive editing level currently in progress.

Exiting the inner recursive edit (such as, with the debugger `c` command) resumes the command running in the next level up. When that command finishes, you can then use `C-M-c` to exit another recursive editing level, and so on. Exiting applies to the innermost level only. Aborting also gets out of only one level of recursive edit; it returns immediately to the command level of the previous recursive edit. If you wish, you can then abort the next recursive editing level.

Alternatively, the command `M-x top-level` aborts all levels of recursive edits, returning immediately to the top level command reader.

The text being edited inside the recursive edit need not be the same text that you were editing at top level. It depends on what the recursive edit is for. If the command that invokes the recursive edit selects a different buffer first, that is the buffer you will edit recursively. In any case, you can switch buffers within the recursive edit in the normal manner (as long as the buffer-switching keys have not been rebound). You could probably do all the rest of your editing inside the recursive edit, visiting files and all. But this could have surprising effects (such as stack overflow) from time to time. So remember to exit or abort the recursive edit when you no longer need it.

In general, we try to minimize the use of recursive editing levels in GNU Emacs. This is because they constrain you to "go back" in a particular order–from the innermost level toward the top level. When possible, we present different activities in separate buffers. Some commands switch to a new major mode but provide a way to switch back. These approaches give you more flexibility to go back to unfinished tasks in the order you choose.

30.10 Dissociated Press

`M-x dissociated-press` is a command for scrambling a file of text either word by word or character by character. Starting from a buffer of straight English, it produces extremely amusing output. The input comes from the current Emacs buffer. Dissociated Press writes its output in a buffer named '`*Dissociation*`', and redisplays that buffer after every couple of lines (approximately) to facilitate reading it.

Dissociated Press asks every so often whether to continue operating. Answer `n` to stop it. You can also stop at any time by typing `C-g`. The dissociation output remains in the '`*Dissociation*`' buffer for you to copy elsewhere if you wish.

Dissociated Press operates by jumping at random from one point in the buffer to another. In order to produce plausible output rather than gibberish, it insists on a certain amount of overlap between the end of one run of consecutive words or characters and the start of the next. That is, if it has just printed out 'president' and then decides to jump to a different point in the file, it might spot the 'ent' in 'pentagon' and continue from there, producing 'presidentagon'.[2] Long sample texts produce the best results.

A positive argument to `M-x dissociated-press` tells it to operate character by character, and specifies the number of overlap characters. A negative argument tells it to operate word by word and specifies the number of overlap words. In this mode, whole words are treated as the elements to be permuted, rather than characters. No argument is equivalent to an argument of two. For your againformation, the output goes only into the buffer '`*Dissociation*`'. The buffer you start with is not changed.

Dissociated Press produces nearly the same results as a Markov chain based on a frequency table constructed from the sample text. It is, however, an independent, ignoriginal invention. Dissociated Press techniquitously copies several consecutive characters from the sample between random choices, whereas a Markov chain would choose randomly for each word or character. This makes for more plausible sounding results, and runs faster.

It is a mustatement that too much use of Dissociated Press can be a developediment to your real work. Sometimes to the point of outragedy. And keep dissociwords out of your documentation, if you want it to be well userenced and properbose. Have fun. Your buggestions are welcome.

[2] This dissociword actually appeared during the Vietnam War, when it was very appropriate.

30.11 Other Amusements

If you are a little bit bored, you can try `M-x hanoi`. If you are considerably bored, give it a numeric argument. If you are very very bored, try an argument of 9. Sit back and watch.

If you want a little more personal involvement, try `M-x gomoku`, which plays the game Go Moku with you.

`M-x blackbox` and `M-x mpuz` are two kinds of puzzles. `blackbox` challenges you to determine the location of objects inside a box by tomography. `mpuz` displays a multiplication puzzle with letters standing for digits in a code that you must guess—to guess a value, type a letter and then the digit you think it stands for.

`M-x dunnet` runs an adventure-style exploration game, which is a bigger sort of puzzle.

When you are frustrated, try the famous Eliza program. Just do `M-x doctor`. End each input by typing `RET` twice.

When you are feeling strange, type `M-x yow`.

30.12 Emulation

GNU Emacs can be programmed to emulate (more or less) most other editors. Standard facilities can emulate these:

EDT (DEC VMS editor)
: Turn on EDT emulation with `M-x edt-emulation-on`. `M-x edt-emulation-off` restores normal Emacs command bindings.

  Most of the EDT emulation commands are keypad keys, and most standard Emacs key bindings are still available. The EDT emulation rebindings are done in the global keymap, so there is no problem switching buffers or major modes while in EDT emulation.

Gosling Emacs
: To turn on emulation of Gosling Emacs (alias Unipress Emacs), type the command `M-x set-gosmacs-bindings`. This redefines many keys, mostly on the `C-x` and `ESC` prefixes, to work as they do in Gosmacs. `M-x set-gnu-bindings` returns to normal GNU Emacs by rebinding the same keys to the definitions they had before you used `M-x set-gosmacs-bindings`.

vi (Berkeley Unix editor)
: Turn on vi emulation with `M-x vi-mode`. This is a major mode that replaces the previously established major mode. All of

the vi commands that, in real vi, enter “input” mode are programmed in the Emacs emulator to return to the previous major mode. Thus, ordinary Emacs serves as vi's “input” mode.

Because vi emulation works through major modes, it does not work to switch buffers during emulation. Return to normal Emacs first.

If you plan to use vi emulation much, you probably want to bind a key to the `vi-mode` command.

vi (alternate emulator)
: Another vi emulator said to resemble real vi more thoroughly is invoked by `M-x vip-mode`. “Input” mode in this emulator is changed from ordinary Emacs so you can use `ESC` to go back to emulated vi command mode. To get from emulated vi command mode back to ordinary Emacs, type `C-z`.

This emulation does not work through major modes, and it is possible to switch buffers in various ways within the emulator. It is not so necessary to assign a key to the command `vip-mode` as it is with `vi-mode` because terminating insert mode does not use it.

For full information, see the long comment at the beginning of the source file, which is '`lisp/vip.el`' in the Emacs distribution.

I am interested in hearing which vi emulator users prefer, as well as in receiving more complete user documentation for either or both emulators. Warning: loading both at once may cause name conflicts; no one has checked.

31 Customization

This chapter talks about various topics relevant to adapting the behavior of Emacs in minor ways. See *The Emacs Lisp Reference Manual* for how to make more far-reaching changes.

All kinds of customization affect only the particular Emacs job that you do them in. They are completely lost when you kill the Emacs job, and have no effect on other Emacs jobs you may run at the same time or later. The only way an Emacs job can affect anything outside of it is by writing a file; in particular, the only way to make a customization 'permanent' is to put something in your '`.emacs`' file or other appropriate file to do the customization in each session. See Section 31.7 [Init File], page 311.

31.1 Minor Modes

Minor modes are optional features which you can turn on or off. For example, Auto Fill mode is a minor mode in which `SPC` breaks lines between words as you type. All the minor modes are independent of each other and of the selected major mode. Most minor modes say in the mode line when they are on; for example, '`Fill`' in the mode line means that Auto Fill mode is on.

Append `-mode` to the name of a minor mode to get the name of a command function that turns the mode on or off. Thus, the command to enable or disable Auto Fill mode is called `M-x auto-fill-mode`. These commands are usually invoked with `M-x`, but you can bind keys to them if you wish. With no argument, the function turns the mode on if it was off and off if it was on. This is known as *toggling*. A positive argument always turns the mode on, and an explicit zero argument or a negative argument always turns it off.

Enabling or disabling some minor modes applies only to the current buffer; each buffer is independent of the other buffers. Therefore, you can enable the mode in particular buffers and disable it in others.

Auto Fill mode allows you to enter filled text without breaking lines explicitly. Emacs inserts newlines as necessary to prevent lines from becoming too long. See Section 21.5 [Filling], page 147.

Outline minor mode provides the same facilities as the major mode called Outline mode; but since it is a minor mode instead, you can combine it with any major mode. See Section 21.8 [Outline Mode], page 152.

Overwrite mode causes ordinary printing characters to replace existing text instead of shoving it over. For example, if the point is in front of the '`B`' in '`FOOBAR`', then in Overwrite mode typing a `G` changes it to '`FOOGAR`', instead of making it '`FOOGBAR`' as usual.

Auto Save mode causes the contents of a buffer to be saved periodically to reduce the amount you will lose in case of a system crash. See Section 15.5 [Auto Save], page 103.

ISO Accents mode makes the characters ‘`’, ‘'’, ‘"’, ‘^’, ‘/’ and ‘~’ combined with the following letter, to produce an accented letter in the ISO Latin-1 character set. See Section 12.4 [European Display], page 71.

The following minor modes normally apply to all buffers at once. Since each is enabled or disabled by the value of a variable, you *can* set them differently for particular buffers, by explicitly making the corresponding variables local in those buffers. See Section 31.2.4 [Locals], page 295.

Abbrev mode allows you to define abbreviations that automatically expand as you type them. For example, ‘`amd`’ might expand to ‘`abbrev mode`’. See Chapter 24 [Abbrevs], page 211, for full information.

Line Number mode enables continuous display in the mode line of the line number of point. See Chapter 2 [Mode Line], page 19.

Scroll Bar mode gives each window a scroll bar (see Section 18.5 [Scroll Bars], page 133). Menu Bar mode gives each frame a menu bar (see Section 18.6 [Menu Bars], page 133). Both of these modes are enabled by default when you use the X Window System.

In Transient Mark mode, every change in the buffer "deactivates" the mark, so that commands that operate on the region will get an error. This means you must either set the mark, or explicitly "reactivate" it, before each command that uses the region. The advantage of Transient Mark mode is that Emacs can display the region highlighted (currently only when using X). See Section 9.1 [Setting Mark], page 51.

31.2 Variables

A *variable* is a Lisp symbol which has a value. The symbol's name is also called the name of the variable. A variable name can contain any characters that can appear in a file, but conventionally variable names consist of words separated by hyphens. A variable can have a documentation string which describes what kind of value it should have and how the value will be used.

Lisp allows any variable to have any kind of value, but most variables that Emacs uses require a value of a certain type. Often the value should always be a string, or should always be a number. Sometimes we say that a certain feature is turned on if a variable is "non-`nil`," meaning that if the variable's value is `nil`, the feature is off, but the feature is on for *any* other value. The conventional value to use to turn on the feature—since you have to pick one particular value when you set the variable—is `t`.

Emacs uses many Lisp variables for internal recordkeeping, as any Lisp program must, but the most interesting variables for you are the ones that exist for the sake of customization. Emacs does not (usually) change the values of these variables; instead, you set the values, and thereby alter and control the behavior of certain Emacs commands. These variables are called *options*. Most options are documented in this manual, and appear in the Variable Index (see [Variable Index], page 395).

One example of a variable which is an option is `fill-column`, which specifies the position of the right margin (as a number of characters from the left margin) to be used by the fill commands (see Section 21.5 [Filling], page 147).

31.2.1 Examining and Setting Variables

`C-h v` *var* `RET`
: Print the value and documentation of variable *var* (`describe-variable`).

`M-x set-variable RET` *var* `RET` *value* `RET`
: Change the value of variable *var* to *value*.

To examine the value of a single variable, use `C-h v` (`describe-variable`), which reads a variable name using the minibuffer, with completion. It prints both the value and the documentation of the variable. For example,

```
C-h v fill-column RET
```

prints something like this:

```
fill-column's value is 75

Documentation:
*Column beyond which automatic line-wrapping should happen.
Automatically becomes buffer-local when set in any fashion.
```

The star at the beginning of the documentation indicates that this variable is an option. `C-h v` is not restricted to options; it allows any variable name.

The most convenient way to set a specific option is with `M-x set-variable`. This reads the variable name with the minibuffer (with completion), and then reads a Lisp expression for the new value using the minibuffer a second time. For example,

```
M-x set-variable RET fill-column RET 75 RET
```

sets `fill-column` to 75.

You can set any variable with a Lisp expression using the function `setq`. Here's how to use it to set `fill-column`:

```
(setq fill-column 75)
```

Setting variables, like all means of customizing Emacs except where explicitly stated, affects only the current Emacs session.

31.2.2 Editing Variable Values

These two functions make it easy to display all the user edit options and change some of them.

`M-x list-options`
: Display a buffer listing names, values and documentation of all options.

`M-x edit-options`
: Change option values by editing a list of options.

`M-x list-options` displays a list of all Emacs option variables, in an Emacs buffer named '`*List Options*`'. Each option is shown with its documentation and its current value. Here is what a portion of it might look like:

```
;; exec-path:
("." "/usr/local/bin" "/usr/ucb" "/bin" "/usr/bin" "/u2/emacs/etc")
*List of directories to search programs to run in subprocesses.
Each element is a string (directory name)
or nil (try the default directory).
;;
;; fill-column:
75
*Column beyond which automatic line-wrapping should happen.
Automatically becomes buffer-local when set in any fashion.
;;
```

`M-x edit-options` goes one step further and immediately selects the '`*List Options*`' buffer; this buffer uses the major mode Options mode, which provides commands that allow you to point at an option and change its value:

`s`
: Set the variable point is in or near to a new value read using the minibuffer.

`x`
: Toggle the variable point is in or near: if the value was `nil`, it becomes `t`; otherwise it becomes `nil`.

`1`
: Set the variable point is in or near to `t`.

`0`
: Set the variable point is in or near to `nil`.

`n`
`p`
: Move to the next or previous variable.

Any changes take effect immediately, and last until you exit from Emacs.

31.2.3 Hooks

A *hook* is a variable where you can store a function or functions to be called on a particular occasion by an existing program. Emacs provides a number of hooks for the sake of customization.

Most of the hooks in Emacs are *normal hooks*. These variables contain lists of functions to be called with no arguments. The reason most hooks are normal hooks is so that you can use them in a uniform way. Every variable in Emacs whose name ends in '`-hook`' is a normal hook.

Most major modes run hooks as the last step of initialization. This makes it easy for a user to customize the behavior of the mode, by overriding the local variable assignments already made by the mode. But hooks may also be used in other contexts. For example, the hook `suspend-hook` runs just before Emacs suspends itself (see Section 3.1 [Exiting], page 23).

The recommended way to add a hook function to a normal hook is by calling `add-hook`. You can use any valid Lisp function as the hook function. For example, here's how to set up a hook to turn on Auto Fill mode when entering Text mode and other modes based on Text mode:

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

The next example shows how to use a hook to customize the indentation of C code. (People often have strong personal preferences for one format compared to another.) Here the hook function is an anonymous lambda expression.

```
(add-hook 'c-mode-hook
  (function (lambda ()
              (setq c-indent-level 4
                    c-argdecl-indent 0
                    c-label-offset -4
                    c-continued-statement-indent 0
                    c-brace-offset 0
                    comment-column 40))))

(setq c++-mode-hook c-mode-hook)
```

It is best to design your hook functions so that the order in which they are executed does not matter. Any dependence on the order is "asking for trouble." However, the order is predictable: the most recently added hook functions are executed first.

31.2.4 Local Variables

`M-x make-local-variable RET var RET`
: Make variable *var* have a local value in the current buffer.

`M-x kill-local-variable RET var RET`
: Make variable *var* use its global value in the current buffer.

`M-x make-variable-buffer-local RET var RET`
: Mark variable *var* so that setting it will make it local to the buffer that is current at that time.

Any variable can be made *local* to a specific Emacs buffer. This means that its value in that buffer is independent of its value in other buffers. A few variables are always local in every buffer. Every other Emacs variable has a *global* value which is in effect in all buffers that have not made the variable local.

`M-x make-local-variable` reads the name of a variable and makes it local to the current buffer. Further changes in this buffer will not affect others, and further changes in the global value will not affect this buffer.

`M-x make-variable-buffer-local` reads the name of a variable and changes the future behavior of the variable so that it will become local automatically when it is set. More precisely, once a variable has been marked in this way, the usual ways of setting the variable automatically do `make-local-variable` first. We call such variables *per-buffer* variables.

Major modes (see Chapter 19 [Major Modes], page 137) always make the variables they set local to the buffer. This is why changing major modes in one buffer has no effect on other buffers. Minor modes also work by setting variables—normally, each minor mode has one controlling variable which is non-`nil` when the mode is enabled (see Section 31.1 [Minor Modes], page 291). For most minor modes, the controlling variable is per buffer.

Emacs contains a number of variables that are always per-buffer. These include `abbrev-mode`, `auto-fill-function`, `case-fold-search`, `comment-column`, `ctl-arrow`, `fill-column`, `fill-prefix`, `indent-tabs-mode`, `left-margin`, `mode-line-format`, `overwrite-mode`, `selective-display-ellipses`, `selective-display`, `tab-width`, and `truncate-lines`. Some other variables are always local in every buffer, but they are used for internal purposes.

`M-x kill-local-variable` reads the name of a variable and makes it cease to be local to the current buffer. The global value of the variable henceforth is in effect in this buffer. Setting the major mode kills all the local variables of the buffer except for a few specially marked variables that are *permanent locals*.

To set the global value of a variable, regardless of whether the variable has a local value in the current buffer, you can use the Lisp function `setq-default`. It works like `setq`. If there is a local value in the current buffer, the local value is not affected by `setq-default`; thus, the new global value may not be visible until you switch to another buffer. For example,

```
(setq-default fill-column 75)
```

`setq-default` is the only way to set the global value of a variable that has been marked with `make-variable-buffer-local`.

Lisp programs can look at a variable's default value with `default-value`. This function takes a symbol as argument and returns its default value. The argument is evaluated; usually you must quote it explicitly. For example,

```
(default-value 'fill-column)
```

31.2.5 Local Variables in Files

A file can specify local variable values for use when you edit the file with Emacs. Visiting the file checks for local variables specifications; it automatically makes these variables local to the buffer, and sets them to the values specified in the file.

There are two ways to specify local variable values: in the first line, or with a local variables list. Here's how to do this with the first line:

`-*- mode: `*modename*`; `*var*`: `*value*`; ... -*-`

You can specify any number of variables/value pairs in this way, each pair with a colon and semicolon as shown above. The major mode should come first, if it is mentioned at all. Here is an example that specifies Lisp mode and sets two variables with numeric values:

```
;; -*-Mode: Lisp; fill-column: 75; comment-column: 50; -*-
```

A *local variables list* goes near the end of the file, in the last page. (It is often best to put it on a page by itself.) The local variables list starts with a line containing the string '`Local Variables:`', and ends with a line containing the string '`End:`'. In between come the variable names and values, one set per line, as '*variable*`: `*value*'. The *value*s are not evaluated; they are used literally.

Here is an example of a local variables list:

```
;;; Local Variables: ***
;;; mode:lisp ***
;;; comment-column:0 ***
;;; comment-start: ";;; "  ***
;;; comment-end:"***" ***
;;; End: ***
```

As you see, each line starts with the prefix '`;;; `' and each line ends with the suffix '` ***`'. Emacs recognizes these as the prefix and suffix based on the first line of the list, by finding them surrounding the magic string '`Local Variables:`'; so it automatically discards them from the other lines of the list.

The usual reason for using a prefix and/or suffix is to embed the local variables list in a comment, so it won't confuse other programs that the file is intended as input for. The example above is for a language where comment lines start with '`;;; `' and end with '`***`'; the local values for `comment-start` and `comment-end` customize the rest of Emacs for this unusual syntax. Don't use a prefix (or a suffix) if you don't need one.

Two "variable names" have special meanings in a local variables list: a value for the variable `mode` really sets the major mode, and a value for the variable `eval` is simply evaluated as an expression and the value is ignored. `mode` and `eval` are not real variables; setting such variables in any other context has no such effect. If `mode` is used in a local variables list, it should be the first entry in the list.

The start of the local variables list must be no more than 3000 characters from the end of the file, and must be in the last page if the file is divided into pages. Otherwise, Emacs will not notice it is there. The purpose of this is so that a stray '`Local Variables:`' not in the last page does not confuse Emacs, and so that visiting a long file that is all one page and has no local variables list need not take the time to search the whole file.

You may be tempted to try to turn on Auto Fill mode with a local variable list. That is a mistake. The choice of Auto Fill mode or not is a matter of individual taste, not a matter of the contents of particular files. If you want to use Auto Fill, set up major mode hooks with your '`.emacs`' file to turn it on (when appropriate) for you alone (see Section 31.7 [Init File], page 311). Don't try to use a local variable list that would impose your taste on everyone.

The variable `enable-local-variables` controls whether to process local variables lists, and thus gives you a chance to override them. Its default value is `t`, which means to process local variables lists. If you set the value to `nil`, Emacs simply ignores local variables lists. Any other value says to query you about each local variables list, showing you the local variables list to consider.

The `eval` "variable" creates special risks, so there is a separate variable `enable-local-eval` to control whether Emacs processes `eval` variables, as well variables with names that end in '`-hook`', '`-hooks`', '`-function`' or '`-functions`'. The three possibilities for the value are `t`, `nil`, and anything else, just as for `enable-local-variables`. The default is `maybe`, which is neither `t` nor `nil`, so normally Emacs queries about `eval` variable settings.

Use the command `normal-mode` to reset the local variables and major mode of a buffer according to the file name and contents, including the local variables list if any. See Section 19.1 [Choosing Modes], page 137.

31.3 Keyboard Macros

A *keyboard macro* is a command defined by the user to abbreviate a sequence of keys. For example, if you discover that you are about to type `C-n C-d` forty times, you can speed your work by defining a keyboard macro to do `C-n C-d` and calling it with a repeat count of forty.

`C-x (`   Start defining a keyboard macro (`start-kbd-macro`).

`C-x )`   End the definition of a keyboard macro (`end-kbd-macro`).

`C-x e` Execute the most recent keyboard macro (`call-last-kbd-macro`).

`C-u C-x (` Re-execute last keyboard macro, then add more keys to its definition.

`C-x q` When this point is reached during macro execution, ask for confirmation (`kbd-macro-query`).

`M-x name-last-kbd-macro`
Give a command name (for the duration of the session) to the most recently defined keyboard macro.

`M-x insert-kbd-macro`
Insert in the buffer a keyboard macro's definition, as Lisp code.

Keyboard macros differ from ordinary Emacs commands in that they are written in the Emacs command language rather than in Lisp. This makes it easier for the novice to write them, and makes them more convenient as temporary hacks. However, the Emacs command language is not powerful enough as a programming language to be useful for writing anything intelligent or general. For such things, Lisp must be used.

You define a keyboard macro while executing the commands which are the definition. Put differently, as you define a keyboard macro, the definition is being executed for the first time. This way, you can see what the effects of your commands are, so that you don't have to figure them out in your head. When you are finished, the keyboard macro is defined and also has been, in effect, executed once. You can then do the whole thing over again by invoking the macro.

31.3.1 Basic Use

To start defining a keyboard macro, type the `C-x (` command (`start-kbd-macro`). From then on, your keys continue to be executed, but also become part of the definition of the macro. '`Def`' appears in the mode line to remind you of what is going on. When you are finished, the `C-x )` command (`end-kbd-macro`) terminates the definition (without becoming part of it!). For example

```
C-x ( M-f foo C-x )
```

defines a macro to move forward a word and then insert '`foo`'.

The macro thus defined can be invoked again with the `C-x e` command (`call-last-kbd-macro`), which may be given a repeat count as a numeric argument to execute the macro many times. `C-x )` can also be given a repeat count as an argument, in which case it repeats the macro that many times right after defining it, but defining the macro counts as the first repetition (since it is executed as you define it). So, giving `C-x )` an argument of 4 executes the macro immediately 3 additional times. An argument of zero to `C-x e` or `C-x )` means repeat the macro indefinitely (until it gets an error or you type `C-g`).

If you wish to repeat an operation at regularly spaced places in the text, define a macro and include as part of the macro the commands to move to the next place you want to use it. For example, if you want to change each line, you should position point at the start of a line, and define a macro to change that line and leave point at the start of the next line. Then repeating the macro will operate on successive lines.

After you have terminated the definition of a keyboard macro, you can add to the end of its definition by typing `C-u C-x (`. This is equivalent to plain `C-x (` followed by retyping the whole definition so far. As a consequence it re-executes the macro as previously defined.

You can use function keys in a keyboard macro, just like keyboard keys. You can even use mouse events, but be careful about that: when the macro replays the mouse event, it uses the original mouse position of that event, the position that the mouse had while you were defining the macro. The effect of this may be hard to predict. (Using the current mouse position would be even less predictable.)

One thing that doesn't always work well in a keyboard macro is the command `C-M-c` (`exit-recursive-edit`). When this command exits a recursive edit that started within the macro, it works as you'd expect. But if it exits a recursive edit that started before you invoked the keyboard macro, it also necessarily exits the keyboard macro as part of the process.

31.3.2 Naming and Saving Keyboard Macros

If you wish to save a keyboard macro for longer than until you define the next one, you must give it a name using `M-x name-last-kbd-macro`. This reads a name as an argument using the minibuffer and defines that name to execute the macro. The macro name is a Lisp symbol, and defining it in this way makes it a valid command name for calling with `M-x` or for binding a key to with `global-set-key` (see Section 31.4.1 [Keymaps], page 302). If you specify a name that has a prior definition other than another keyboard macro, an error message is printed and nothing is changed.

Once a macro has a command name, you can save its definition in a file. Then it can be used in another editing session. First, visit the file you want to save the definition in. Then use this command:

`M-x insert-kbd-macro RET` *macroname* `RET`

This inserts some Lisp code that, when executed later, will define the same macro with the same definition it has now. (You need not understand Lisp code to do this, because `insert-kbd-macro` writes the Lisp code for you.)

Then save the file. You can load the file later with `load-file` (see Section 23.4 [Lisp Libraries], page 205). If the file you save in is your init file '`~/.emacs`' (see Section 31.7 [Init File], page 311) then the macro will be defined each time you run Emacs.

If you give `insert-kbd-macro` a prefix argument, it makes additional Lisp code to record the keys (if any) that you have bound to the keyboard macro, so that the macro will be reassigned the same keys when you load the file.

31.3.3 Executing Macros with Variations

Using `C-x q` (`kbd-macro-query`), you can get an effect similar to that of `query-replace`, where the macro asks you each time around whether to make a change. When you are defining the macro, type `C-x q` at the point where you want the query to occur. During macro definition, the `C-x q` does nothing, but when you run the macro later, `C-x q` asks you interactively whether to continue.

The valid responses when `C-x q` asks are `SPC` (or `y`), `DEL` (or `n`), `ESC` (or `q`), `C-l` and `C-r`. The answers are the same as in query replace, though not all of the query-replace options are meaningful.

Specifically, `SPC` means to continue. `DEL` means to skip the remainder of this repetition of the macro and start right away with the next repetition. `ESC` means to skip the remainder of this repetition and cancel further repetition. `C-l` redraws the screen and asks you again for a character to say what to do.

`C-r` enters a recursive editing level, in which you can perform editing which is not part of the macro. When you exit the recursive edit using `C-M-c`, you are asked again how to continue with the keyboard macro. If you type a `SPC` at this time, the rest of the macro definition is executed. It is up to you to leave point and the text in a state such that the rest of the macro will do what you want.

`C-u C-x q`, which is `C-x q` with a numeric argument, performs a different function. It enters a recursive edit reading input from the keyboard, both when you type it during the definition of the macro, and when it is executed from the macro. During definition, the editing you do inside the recursive edit does not become part of the macro. During macro execution, the recursive edit gives you a chance to do some particularized editing. See Section 30.9 [Recursive Edit], page 287.

31.4 Customizing Key Bindings

This section describes *key bindings* which map keys to commands, and the *keymaps* which record key bindings. It also explains how to customize key bindings.

Recall that a command is a Lisp function whose definition provides for interactive use. Like every Lisp function, a command has a function name which usually consists of lower case letters and hyphens.

31.4.1 Keymaps

The bindings between key sequences and command functions are recorded in data structures called *keymaps*. Emacs has many of these, each used on particular occasions.

Recall that a *key sequence* (*key*, for short) is a sequence of *input events* that have a meaning as a unit. Input events include characters, function keys and mouse buttons—all the inputs that you can send to the computer with your terminal. A key sequence gets its meaning from its *binding*, which says what command it runs. The function of keymaps is to record these bindings.

The *global* keymap is the most important keymap because it is always in effect. The global keymap defines keys for Fundamental mode; most of these definitions are common to all (or most) major modes. Each major or minor mode can have its own keymap which overrides the definitions of some keys.

For example, a self-inserting character such as `g` is self-inserting because the global keymap binds it to the command `self-insert-command`. The standard Emacs editing characters such as `C-a` also get their standard meanings from the global keymap. Commands to rebind keys, such as `M-x global-set-key`, actually work by storing the new binding in the proper place in the global map. See Section 31.4.5 [Rebinding], page 305.

Meta characters work differently; Emacs translates each Meta character into a pair of characters starting with `ESC`. When you type the character `M-a` in a key sequence, Emacs replaces it with `ESC a`. A meta key comes in as a single input event, but becomes two events for purposes of key bindings. The reason for this is historical, and we might change it someday.

Most modern keyboards have function keys as well as character keys. Function keys send input events just as character keys do, and keymaps can have bindings for them.

On many terminals, typing a function key actually sends the computer a sequence of characters; the precise details of the sequence depends on which function key and on the model of terminal you are using. (Often he sequence starts with `ESC [`.) If Emacs understands your terminal type properly, it recognizes the character sequences forming function keys wherever they occur in a key sequence (not just at the beginning). Thus, for most purposes, you can pretend the function keys reach Emacs directly and ignore their encoding as character sequences.

Mouse buttons also produce input events. These events come with other data—the window and position where you pressed or released the button, and a timestamp. But only the choice of button matters for key bindings; the other data matters only if a command looks at it. (Commands designed for mouse invocation usually do look at the other data.)

A keymap records definitions for single events. Interpreting a key sequence of multiple events involves a chain of keymaps. The first keymap gives a definition for the first event; this definition is another keymap, which is used to look up the second event in the sequence, and so on.

Key sequences can mix function keys and characters. For example, `C-x SELECT` makes sense. If you make `SELECT` a prefix key, then `SELECT C-n` makes sense. You can even mix mouse events with keyboard events, but we recommend against it, because such sequences are inconvenient to type in.

31.4.2 Prefix Keymaps

A prefix key such as `C-x` or `ESC` has its own keymap, which holds the definition for the event that immediately follows that prefix.

The definition of a prefix key is the keymap to use for looking up the following event. The definition can also be a Lisp symbol whose function definition is the following keymap; the effect is the same, but it provides a command name for the prefix key that can be used as a description of what the prefix key is for. Thus, the binding of `C-x` is the symbol `Ctl-X-Prefix`, whose function definition is the keymap for `C-x` commands. The definitions of `C-c`, `C-x`, `C-h` and `ESC` as prefix keys appear in the global map, so these prefix keys are always available.

Some prefix keymaps are stored in variables with names:

- `ctl-x-map` is the variable name for the map used for characters that follow `C-x`.
- `help-map` is for characters that follow `C-h`.
- `esc-map` is for characters that follow `ESC`. Thus, all Meta characters are actually defined by this map.
- `ctl-x-4-map` is for characters that follow `C-x 4`.
- `mode-specific-map` is for characters that follow `C-c`.

31.4.3 Local Keymaps

So far we have explained the ins and outs of the global map. Major modes customize Emacs by providing their own key bindings in *local keymaps*. For example, C mode overrides `TAB` to make it indent the current line for C code. Portions of text in the buffer can specify their own keymaps to substitute for the keymap of the buffer's major mode.

Minor modes can also have local keymaps. Whenever a minor mode is in effect, the definitions in its keymap override both the major mode's local keymap and the global keymap.

The local keymaps for Lisp mode, C mode, and several other major modes always exist even when not in use. These are kept in variables named `lisp-mode-map`, `c-mode-map`, and so on. For major modes less often used, the local keymap is normally constructed only when the mode is used for the first time in a session. This is to save space.

All minor mode keymaps are created in advance. There is no way to defer their creation until the minor mode is enabled.

A local keymap can locally redefine a key as a prefix key by defining it as a prefix keymap. If the key is also defined globally as a prefix, then its local and global definitions (both keymaps) effectively combine: both of them are used to look up the event that follows the prefix key. Thus, if the mode's local keymap defines `C-c` as another keymap, and that keymap defines `C-z` as a command, this provides a local meaning for `C-c C-z`. This does not affect other sequences that start with `C-c`; If those sequences don't have their own local bindings, their global bindings remain in effect.

Another way to think of this is that Emacs handles a multi-event key sequence by looking in several keymaps, one by one, for a binding of the whole key sequence. First it checks the minor mode keymaps for minor modes that are enabled, then it checks the major mode's keymap, and then it checks the global keymap. This is not precisely how key lookup works, but it's good enough for understanding ordinary circumstances.

31.4.4 Minibuffer Keymaps

The minibuffer has its own set of local keymaps; they contain various completion and exit commands.

- `minibuffer-local-map` is used for ordinary input (no completion).
- `minibuffer-local-ns-map` is similar, except that `SPC` exits just like `RET`. This is used mainly for Mocklisp compatibility.
- `minibuffer-local-completion-map` is for permissive completion.
- `minibuffer-local-must-match-map` is for strict completion and for cautious completion.

31.4.5 Changing Key Bindings Interactively

The way to redefine an Emacs key is to change its entry in a keymap. You can change the global keymap, in which case the change is effective in all major modes (except those that have their own overriding local definitions for the same key). Or you can change the current buffer's local map, which affects all buffers using the same major mode.

`M-x global-set-key RET` *key cmd* `RET`
: Define *key* globally to run *cmd*.

`M-x local-set-key RET` *key cmd* `RET`
: Define *key* locally (in the major mode now in effect) to run *cmd*.

`M-x global-unset-key RET` *key*
: Make *key* undefined in the global map.

`M-x local-unset-key RET` *key*
: Make *key* undefined locally (in the major mode now in effect).

For example, suppose you like to execute commands in a subshell within an Emacs buffer, instead of suspending Emacs and executing commands in your login shell. Normally, `C-z` is bound to the function `suspend-emacs`, but you can change `C-z` to invoke an interactive subshell within Emacs, by binding it to `shell` as follows:

```
M-x global-set-key RET C-z shell RET
```

`global-set-key` reads the command name after the key. After you press the key, a message like this appears so that you can confirm that you are binding the key you want:

```
Set key C-z to command:
```

You can redefine function keys and mouse events in the same way; just type the function key or click the mouse when it's time to specify the key to rebind.

You can rebind a key that contains more than one event in the same way. Emacs keeps reading the key to rebind until it is a complete key (that is, not a prefix key). Thus, if you type `C-f` for *key*, that's the end; the minibuffer is entered immediately to read *cmd*. But if you type `C-x`, another character is read; if that is `4`, another character is read, and so on. For example,

```
M-x global-set-key RET C-x 4 $ spell-other-window RET
```

would redefine `C-x 4 $` to run the (fictitious) command `spell-other-window`.

You can remove the global definition of a key with `global-unset-key`. This makes the key *undefined*; if you type it, Emacs will just beep. Similarly, `local-unset-key` makes a key undefined in the current major mode keymap, which makes the global definition (or lack of one) come back into effect in that major mode.

If you have redefined (or undefined) a key and you subsequently wish to retract the change, undefining the key will not do the job—you need to redefine the key with its standard definition. To find the name of the standard definition of a key, go to a Fundamental mode buffer and use `C-h c`. The documentation of keys in this manual also lists their command names.

If you want to prevent yourself from invoking a command by mistake, it is better to disable the command than to undefine the key. A disabled command is less work to invoke when you really want to. See Section 31.4.9 [Disabling], page 310.

31.4.6 Rebinding Keys in Your Init File

If you have a set of default definitions that you like to use all the time, you can put commands in your '`.emacs`' file by using their Lisp syntax. Thus, the first `global-set-key` command in this section could be put in an '`.emacs`' file in either of the two following formats:

```
(global-set-key "\C-z" 'shell)
```

or as:

```
(global-set-key [?\C-z] 'shell)
```

When the key sequence consists of characters, like this one, you can write it as a string or as a vector. The first format specifies the key sequence as a string, `"\C-z"`. The second format uses a vector to specify the key sequence. The square brackets ('`[. . .]`') delimit the contents of the vector. The vector in this example contains just one element, which is the integer code corresponding to `C-z`. The question mark is the Lisp syntax for a character constant; the character must follow with no intervening spaces.

The single-quote before `shell` marks it as a constant symbol rather than a variable. If you omit the quote, Emacs tries to evaluate `shell` immediately as a variable. This probably causes an error; it certainly isn't what you want.

Here is another example that binds a key sequence two characters long:

```
(global-set-key "\C-xl" 'make-symbolic-link)
```

or:

```
(global-set-key [?\C-x ?l] 'make-symbolic-link)
```

31.4.7 Rebinding Function Keys

Key sequences can contain function keys as well as ordinary characters. Just as Lisp characters (actually integers) represent keyboard characters, Lisp symbols represent function keys. If the function key has a word as its label, then that word is also the name of the corresponding Lisp symbol. Here are the conventional Lisp names for other function keys:

`left`, `up`, `right`, `down`
: Cursor arrow keys.

`begin`, `end`, `home`, `next`, `prior`
: Other cursor repositioning keys.

`select`, `print`, `execute`, `backtab`
`insert`, `undo`, `redo`, `clearline`
`insertline`, `deleteline`, `insertchar`, `deletechar`,
: Miscellaneous function keys.

`f1` ... `f35`
: Numbered function keys (across the top of the keyboard).

`kp-add`, `kp-subtract`, `kp-multiply`, `kp-divide`
`kp-backtab`, `kp-space`, `kp-tab`, `kp-enter`
`kp-separator`, `kp-decimal`, `kp-equal`
: Keypad keys (to the right of the regular keyboard), with names or punctuation.

`kp-0`, `kp-1`, ... `kp-9`
: Keypad keys with digits.

`kp-f1`, `kp-f2`, `kp-f3`, `kp-f4`
: Keypad PF keys.

These names are conventional, but some systems (especially when using X windows) may use different names. To make certain what symbol is used for a given function key on your terminal, type `C-h c` followed by that key.

A key sequence which contains non-characters must be a vector rather than a string. To write a vector, write square brackets containing the vector elements. Write spaces to separate the elements. If an element is a symbol, simply write the symbol's name—no delimiters or punctuation are needed. If an element is a character, write a Lisp character constant, which is '`?`' followed by the character as it would appear in a string.

Thus, to bind function key '`f1`' to the command `rmail`, write the following:

```
(global-set-key [f1] 'rmail)
```

To bind the right-arrow key to the command `forward-char`, you can use this expression:

```
(global-set-key [right] 'forward-char)
```

using the Lisp syntax for a vector containing the symbol `right`. (This binding is present in Emacs by default.)

You can mix function keys and characters in a key sequence. This example binds `C-x RIGHT` to the command `forward-page`.

```
(global-set-key [?\C-x right] 'forward-page)
```

where `?\C-x` is the Lisp character constant for the character `C-x`. The vector element `right` is a symbol and therefore does not take a question mark.

You can use the modifier keys `CTRL`, `META`, `HYPER`, `SUPER`, `ALT` and `SHIFT` with function keys. To represent these modifiers, prepend the strings '`C-`', '`M-`', '`H-`', '`s-`', '`A-`' and '`S-`' to the symbol name. Thus, here is how to make `Hyper-Meta-RIGHT` move forward a word:

```
(global-set-key [H-M-right] 'forward-word)
```

31.4.8 Rebinding Mouse Buttons

Emacs uses Lisp symbols to designate mouse buttons, too. The ordinary mouse events in Emacs are *click* events; these happen when you press a button and release it without moving the mouse. You can also get *drag* events, when you move the mouse while holding the button down. Drag events happen when you finally let go of the button.

The symbols for basic click events are `mouse-1` for the leftmost button, `mouse-2` for the next, and so on. Here is how you can redefine the second mouse button to split the current window:

```
(global-set-key [mouse-2] 'split-window-vertically)
```

The symbols for drag events are similar, but have the prefix '`drag-`' before the word '`mouse`'. For example, dragging the left button generates a `drag-mouse-1` event.

You can also define events that occur when a mouse button is pressed down. These events start with '`down-`' instead of '`drag-`'. Such events are generated only if they have key bindings. When you get a button-down event, a corresponding click or drag event will always follow.

If you wish, you can distinguish single, double, and triple clicks. A double click means clicking a mouse button twice in approximately the same place. The first click generates an ordinary click event. The second click, if it comes soon enough, generates a double-click event instead. The event type for a double click event starts with '`double-`': for example, '`double-mouse-3`'.

This means that you can give a special meaning to the second click at the same place, but it must act on the assumption that the ordinary single click definition has run when the first click was received.

This constrains what you can do with double clicks, but user interface designers say that this constraint ought to be followed in any case. A double click should do something similar to the single click, only "more so". The command for the double-click event should perform the extra work for the double click.

If a double-click event has no binding, it changes to the corresponding single-click event. Thus, if you don't define a particular double click specially, it executes the single-click command twice.

Emacs also supports triple-click events whose names start with '`triple-`'. Emacs does not distinguish quadruple clicks as event types; clicks beyond the third generate additional triple-click events. However, the full number of clicks is recorded in the event list, so you can distinguish if you really want to. We don't recommend distinct meanings for more than three clicks, but sometimes it is useful for subsequent clicks to cycle through the same set of three meanings, so that four clicks are equivalent to one click, five are equivalent to two, and six are equivalent to three.

Emacs also records multiple presses in drag and button-down events. For example, when you press a button twice, then move the mouse while holding the button, Emacs gets a '`double-drag-`' event. And at the moment when you press it down for the second time, Emacs gets a '`double-down-`' event (which is ignored, like all button-down events, if it has no binding).

The variable `double-click-time` specifies how long may elapse between clicks that are recognized as a pair. Its value is measured in milliseconds. If the value is `nil`, double clicks are not detected at all. If the value is `t`, then there is no time limit.

The symbols for mouse events also indicate the status of the modifier keys, with the usual prefixes '`C-`', '`M-`', '`H-`', '`s-`', '`A-`' and '`S-`'. These always precede '`double-`' or '`triple-`', which always precede '`drag-`' or '`down-`'.

A frame includes areas that don't show text from the buffer, such as the mode line and the scroll bar. You can tell whether a mouse button comes from a special area of the screen by means of dummy "prefix keys." For example, if you click the mouse in the mode line, you get the prefix key `mode-line` before the ordinary mouse-button symbol. Thus, here is how to define the command for clicking the left button in a mode line to run `scroll-up`:

```
(global-set-key [mode-line mouse-1] 'scroll-up)
```

Here is the complete list of these dummy prefix keys and their meanings:

`mode-line`
: The mouse was in the mode line of a window.

`vertical-line`
: The mouse was in the vertical line separating side-by-side windows.

`vertical-scroll-bar`
: The mouse was in a horizontal scroll bar. This is the kind of scroll bar people normally use.

`horizontal-scroll-bar`
: The mouse was in a horizontal scroll bar. Horizontal scroll bars do horizontal scrolling, and people don't use them often.

You can put more than one mouse button in a key sequence, but it isn't usual to do so.

31.4.9 Disabling Commands

Disabling a command marks the command as requiring confirmation before it can be executed. The purpose of disabling a command is to prevent beginning users from executing it by accident and being confused.

Attempting to invoke a disabled command interactively in Emacs causes the display of a window containing the command's name, its documentation, and some instructions on what to do immediately; then Emacs asks for input saying whether to execute the command as requested, enable it and execute, or cancel it. If you decide to enable the command, you are asked whether to do this permanently or just for the current session. Enabling permanently works by automatically editing your '.emacs' file.

The direct mechanism for disabling a command is to have a non-nil disabled property on the Lisp symbol for the command. Here is the Lisp program to do this:

```
(put 'delete-region 'disabled t)
```

If the value of the disabled property is a string, that string is included in the message printed when the command is used:

```
(put 'delete-region 'disabled
     "Text deleted this way cannot be yanked back!\n")
```

You can make a command disabled either by editing the '.emacs' file directly or with the command M-x disable-command, which edits the '.emacs' file for you. Likewise, M-x enable-command edits '.emacs' to enable a command permanently. See Section 31.7 [Init File], page 311.

Whether a command is disabled is independent of what key is used to invoke it; it also applies if the command is invoked using M-x. Disabling a command has no effect on calling it as a function from Lisp programs.

31.5 Keyboard Translations

Some keyboards do not make it convenient to send all the special characters that Emacs uses. The most common problem case is the DEL character. Some keyboards provide no convenient way to type this very important character—usually because they were designed to expect the character C-h to be used for deletion. On these keyboard, if you press the key normally used for deletion, Emacs handles the C-h as a prefix character and offers you a list of help options, which is not what you want.

You can work around this problem within Emacs by setting up keyboard translations to turn C-h into DEL and DEL into C-h, as follows:

```
;; Translate C-h to DEL.
(keyboard-translate ?\C-h ?\C-?)
;; Translate DEL to C-h.
(keyboard-translate ?\C-? ?\C-h)
```

Keyboard translations are not the same as key bindings in keymaps (see Section 31.4.1 [Keymaps], page 302). Emacs contains numerous keymaps that apply in different situations, but there is only one set of keyboard translations, and it applies to every character that Emacs reads from the terminal. Keyboard translations take place at the lowest level of input processing; the keys that are looked up in keymaps contain the characters that result from keyboard translation.

For full information about how to use keyboard translations, see section "Translating Input" in *The Emacs Lisp Reference Manual*.

31.6 The Syntax Table

All the Emacs commands which parse words or balance parentheses are controlled by the *syntax table*. The syntax table says which characters are opening delimiters, which are parts of words, which are string quotes, and so on. Each major mode has its own syntax table (though sometimes related major modes use the same one) which it installs in each buffer that uses that major mode. The syntax table installed in the current buffer is the one that all commands use, so we call it "the" syntax table. A syntax table is a Lisp object, a vector of length 256 whose elements are numbers.

To display a description of the contents of the current syntax table, type `C-h s` (`describe-syntax`). The description of each character includes both the string you would have to give to `modify-syntax-entry` to set up that character's current syntax, and some English to explain that string if necessary.

For full information on the syntax table, see section "Syntax Table" in *The Emacs Lisp Reference Manual*.

31.7 The Init File, '~/.emacs'

When Emacs is started, it normally loads a Lisp program from the file '`.emacs`' in your home directory. We call this file your *init file* because it specifies how to initialize Emacs for you. You can use the command line switches '`-q`' and '`-u`' to tell Emacs whether to load an init file, and which one (see Chapter 3 [Entering Emacs], page 23).

There can also be a *default init file*, which is the library named '`default.el`', found via the standard search path for libraries. The Emacs distribution contains no such library; your site may create one for local customizations. If this library exists, it is loaded whenever you start Emacs (except when you specify '-q'). But your init file, if any, is loaded first; if it sets `inhibit-default-init` non-`nil`, then '`default`' is not loaded.

If you have a large amount of code in your '`.emacs`' file, you should move it into another file such as '`~/`*something*`.el`', byte-compile it, and make your '`.emacs`' file load it with `(load "~/`*something*`")`. See section "Byte Compilation" in *the Emacs Lisp Reference Manual*, for more information about compiling Emacs Lisp programs.

31.7.1 Init File Syntax

The '`.emacs`' file contains one or more Lisp function call expressions. Each of these consists of a function name followed by arguments, all surrounded by parentheses. For example, `(setq fill-column 60)` calls the function `setq` to set the variable `fill-column` (see Section 21.5 [Filling], page 147) to 60.

The second argument to `setq` is an expression for the new value of the variable. This can be a constant, a variable, or a function call expression. In '`.emacs`', constants are used most of the time. They can be:

Numbers: Numbers are written in decimal, with an optional initial minus sign.

Strings: Lisp string syntax is the same as C string syntax with a few extra features. Use a double-quote character to begin and end a string constant.

In a string, you can include newlines and special characters literally. But often it is cleaner to use backslash sequences for them: '`\n`' for newline, '`\b`' for backspace, '`\r`' for carriage return, '`\t`' for tab, '`\f`' for formfeed (control-L), '`\e`' for escape, '`\\`' for a backslash, '`\"`' for a double-quote, or '`\`*ooo*' for the character whose octal code is *ooo*. Backslash and double-quote are the only characters for which backslash sequences are mandatory.

'`\C-`' can be used as a prefix for a control character, as in '`\C-s`' for ASCII control-S, and '`\M-`' can be used as a prefix for a Meta character, as in '`\M-a`' for `Meta-A` or '`\M-\C-a`' for `Control-Meta-A`.

Characters:

Lisp character constant syntax consists of a '`?`' followed by either a character or an escape sequence starting with '`\`'. Examples: `?x`, `?\n`, `?\"`, `?\)`. Note that strings and characters are not interchangeable in Lisp; some contexts require one and some contexts require the other.

True: `t` stands for 'true'.

False: `nil` stands for 'false'.

Other Lisp objects:
Write a single-quote (') followed by the Lisp object you want.

31.7.2 Init File Examples

Here are some examples of doing certain commonly desired things with Lisp expressions:

- Make `TAB` in C mode just insert a tab if point is in the middle of a line.

  ```
  (setq c-tab-always-indent nil)
  ```

  Here we have a variable whose value is normally `t` for 'true' and the alternative is `nil` for 'false'.

- Make searches case sensitive by default (in all buffers that do not override this).

  ```
  (setq-default case-fold-search nil)
  ```

  This sets the default value, which is effective in all buffers that do not have local values for the variable. Setting `case-fold-search` with `setq` affects only the current buffer's local value, which is not what you probably want to do in an init file.

- Make Text mode the default mode for new buffers.

  ```
  (setq default-major-mode 'text-mode)
  ```

  Note that `text-mode` is used because it is the command for entering Text mode. The single-quote before it makes the symbol a constant; otherwise, `text-mode` would be treated as a variable name.

- Turn on Auto Fill mode automatically in Text mode and related modes.

  ```
  (add-hook 'text-mode-hook
    '(lambda () (auto-fill-mode 1)))
  ```

  This shows how to add a hook function to a normal hook variable (see Section 31.2.3 [Hooks], page 294). The function we supply is a list starting with `lambda`, with a single-quote in front of it to make it a list constant rather than an expression.

  It's beyond the scope of this manual to explain Lisp functions, but for this example it is enough to know that the effect is to execute `(auto-fill-mode 1)` when Text mode is entered. You can replace it with any other expression that you like, or with several expressions in a row.

  Emacs comes with a function named `turn-on-auto-fill` whose definition is `(lambda () (auto-fill-mode 1))`. Thus, a simpler way to write the above example is as follows:

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

- Load the installed Lisp library named '`foo`' (actually a file '`foo.elc`' or '`foo.el`' in a standard Emacs directory).

  ```
  (load "foo")
  ```

  When the argument to `load` is a relative file name, not starting with '`/`' or '`~`', `load` searches the directories in `load-path` (see Section 23.4 [Lisp Libraries], page 205).
- Load the compiled Lisp file '`foo.elc`' from your home directory.

  ```
  (load "~/foo.elc")
  ```

  Here an absolute file name is used, so no searching is done.
- Rebind the key `C-x l` to run the function `make-symbolic-link`.

  ```
  (global-set-key "\C-xl" 'make-symbolic-link)
  ```

  or

  ```
  (define-key global-map "\C-xl" 'make-symbolic-link)
  ```

  Note once again the single-quote used to refer to the symbol `make-symbolic-link` instead of its value as a variable.
- Do the same thing for C mode only.

  ```
  (define-key c-mode-map "\C-xl" 'make-symbolic-link)
  ```

- Redefine all keys which now run `next-line` in Fundamental mode so that they run `forward-line` instead.

  ```
  (substitute-key-definition 'next-line 'forward-line
                             global-map)
  ```

- Make `C-x C-v` undefined.

  ```
  (global-unset-key "\C-x\C-v")
  ```

  One reason to undefine a key is so that you can make it a prefix. Simply defining `C-x C-v` *anything* will make `C-x C-v` a prefix, but `C-x C-v` must first be freed of its usual non-prefix definition.
- Make '`$`' have the syntax of punctuation in Text mode. Note the use of a character constant for '`$`'.

  ```
  (modify-syntax-entry ?\$ "." text-mode-syntax-table)
  ```

- Enable the use of the command `eval-expression` without confirmation.

  ```
  (put 'eval-expression 'disabled nil)
  ```

31.7.3 Terminal-specific Initialization

Each terminal type can have a Lisp library to be loaded into Emacs when it is run on that type of terminal. For a terminal type named *termtype*, the library is called '`term/`*termtype*' and it is found by searching the directories `load-path` as usual and trying the suffixes '`.elc`' and '`.el`'. Normally it appears in the subdirectory '`term`' of the directory where most Emacs libraries are kept.

The usual purpose of the terminal-specific library is to define the escape sequences used by the terminal's function keys using the library '`keypad.el`'. See the file '`term/vt100.el`' for an example of how this is done.

When the terminal type contains a hyphen, only the part of the name before the first hyphen is significant in choosing the library name. Thus, terminal types '`aaa-48`' and '`aaa-30-rv`' both use the library '`term/aaa`'. The code in the library can use `(getenv "TERM")` to find the full terminal type name.

The library's name is constructed by concatenating the value of the variable `term-file-prefix` and the terminal type. Your '`.emacs`' file can prevent the loading of the terminal-specific library by setting `term-file-prefix` to `nil`.

Emacs runs the hook `term-setup-hook` at the end of initialization, after both your '`.emacs`' file and any terminal-specific library have been read in. Add hook functions to this hook if you wish to override part of any of the terminal-specific libraries and to define initializations for terminals that do not have a library. See Section 31.2.3 [Hooks], page 294.

31.7.4 How Emacs Finds Your Init File

Normally Emacs uses the environment variable `HOME` to find '`.emacs`'; that's what '`~`' means in a file name. But if you have done `su`, Emacs tries to find your own '`.emacs`', not that of the user you are currently pretending to be. The idea is that you should get your own editor customizations even if you are running as the super user.

More precisely, Emacs first determines which user's init file to use. It gets the user name from the environment variables `USER` and `LOGNAME`; if neither of those exists, it uses effective user-ID. If that user name matches the real user-ID, then Emacs uses `HOME`; otherwise, it looks up the home directory corresponding to that user name in the system's data base of users.

32 Correcting Mistakes (Yours or Emacs's)

If you type an Emacs command you did not intend, the results are often mysterious. This chapter tells what you can do to cancel your mistake or recover from a mysterious situation. Emacs bugs and system crashes are also considered.

32.1 Quitting and Aborting

`C-g` Quit. Cancel running or partially typed command.

`C-]` Abort innermost recursive editing level and cancel the command which invoked it (`abort-recursive-edit`).

`M-x top-level`
Abort all recursive editing levels that are currently executing.

`C-x u` Cancel an already-executed command, usually (`undo`).

There are two ways of cancelling commands which are not finished executing: *quitting* with `C-g`, and *aborting* with `C-]` or `M-x top-level`. Quitting 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 30.9 [Recursive Edit], page 287.)

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.

`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 32.2.6 [Emergency Escape], page 320.

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 `M-x top-level` is equivalent to "enough" `C-]` commands to get you out of all the levels of recursive edits that you are in. `C-]` gets you out one level at a time, but `M-x top-level` goes out all levels at once. Both `C-]` and `M-x top-level` are like all other commands, and unlike `C-g`, in that they are effective only when Emacs is ready for a command. `C-]` is an ordinary key and has its meaning only because of its binding in the keymap. See Section 30.9 [Recursive Edit], page 287.

`C-x u` (`undo`) is not strictly speaking a way of cancelling a command, but you can think of it as cancelling a command already finished executing. See Chapter 5 [Undo], page 33.

32.2 Dealing with Emacs Trouble

This section describes various conditions in which Emacs fails to work normally, and how to recognize them and correct them.

32.2.1 If `DEL` Fails to Delete

If you find that `DEL` enters Help like `Control-h` instead of deleting a character, your terminal is sending the wrong code for `DEL`. You can work around this problem by changing the keyboard translation table (see Section 31.5 [Keyboard Translations], page 310).

32.2.2 Recursive Editing Levels

Recursive editing levels are important and useful features of Emacs, but they can seem like malfunctions to the user who does not understand them.

If the mode line has square brackets '`[. . .]`' around the parentheses that contain the names of the major and minor modes, you have entered a recursive editing level. If you did not do this on purpose, or if you don't understand what that means, you should just get out of the recursive editing level. To do so, type `M-x top-level`. This is called getting back to top level. See Section 30.9 [Recursive Edit], page 287.

32.2.3 Garbage on the Screen

If the data on the screen looks wrong, the first thing to do is see whether the text is really wrong. Type `C-l`, to redisplay the entire screen. If the screen appears correct after this, the problem was entirely in the previous screen update.

Display updating problems often result from an incorrect termcap entry for the terminal you are using. The file '`etc/TERMS`' in the Emacs distribution gives the fixes for known problems of this sort. '`INSTALL`' contains general advice for these problems in one of its sections. Very likely there is simply insufficient padding for certain display operations. To investigate the possibility that you have this sort of problem, try Emacs on another terminal made by a different manufacturer. If problems happen frequently on one kind of terminal but not another kind, it is likely to be a bad termcap entry, though it could also be due to a bug in Emacs that appears for terminals that have or that lack specific features.

32.2.4 Garbage in the Text

If `C-l` shows that the text is wrong, try undoing the changes to it using `C-x u` until it gets back to a state you consider correct. Also try `C-h l` to find out what command you typed to produce the observed results.

If a large portion of text appears to be missing at the beginning or end of the buffer, check for the word '`Narrow`' in the mode line. If it appears, the text is still present, but temporarily off-limits. To make it accessible again, type `C-x n w`. See Section 30.4 [Narrowing], page 282.

32.2.5 Spontaneous Entry to Incremental Search

If Emacs spontaneously displays '`I-search:`' at the bottom of the screen, it means that the terminal is sending `C-s` and `C-q` according to the poorly designed xon/xoff "flow control" protocol.

If this happens to you, your best recourse is to put the terminal in a mode where it will not use flow control, or give it so much padding that it will never send a `C-s`. (One way to increase the amount of padding is to set the variable `baud-rate` to a larger value. Its value is the terminal output speed, measured in the conventional units of baud.)

If you don't succeed in turning off flow control, the next best thing is to tell Emacs to cope with it. To do this, call the function `enable-flow-control`.

Typically there are particular terminal types with which you must use flow control. You can conveniently ask for flow control on those terminal types only, using `enable-flow-control-on`. For example, if you find you must use flow control on VT-100 and H19 terminals, put the following in your '.emacs' file:

```
(enable-flow-control-on "vt100" "h19")
```

When flow control is enabled, you must type `C-\` to get the effect of a `C-s`, and type `C-^` to get the effect of a `C-q`. (These aliases work by means of keyboard translations; see Section 31.5 [Keyboard Translations], page 310.)

32.2.6 Emergency Escape

Because at times there have been bugs causing Emacs to loop without checking `quit-flag`, a special feature causes Emacs to be suspended immediately if you type a second `C-g` while the flag is already set, so you can always get out of GNU Emacs. Normally Emacs recognizes and clears `quit-flag` (and quits!) quickly enough to prevent this from happening.

When you resume Emacs after a suspension caused by multiple `C-g`, it asks two questions before going back to what it had been doing:

```
Auto-save? (y or n)
Abort (and dump core)? (y or n)
```

Answer each one with `y` or `n` followed by `RET`.

Saying `y` to '`Auto-save?`' causes immediate auto-saving of all modified buffers in which auto-saving is enabled.

Saying `y` to '`Abort (and dump core)?`' causes an illegal instruction to be executed, dumping core. This is to enable a wizard to figure out why Emacs was failing to quit in the first place. Execution does not continue after a core dump. If you answer `n`, execution does continue. With luck, GNU Emacs will ultimately check `quit-flag` and quit normally. If not, and you type another `C-g`, it is suspended again.

If Emacs is not really hung, just slow, you may invoke the double `C-g` feature without really meaning to. Then just resume and answer `n` to both questions, and you will arrive at your former state. Presumably the quit you requested will happen soon.

The double-`C-g` feature is turned off when Emacs is running under the X Window System, since the you can use the window manager to kill Emacs or to create another window and run another program.

32.2.7 Help for Total Frustration

If using Emacs (or something else) becomes terribly frustrating and none of the techniques described above solve the problem, Emacs can still help you.

First, if the Emacs you are using is not responding to commands, type `C-g C-g` to get out of it and then start a new one.

Second, type `M-x doctor RET`.

The doctor will help you feel better. Each time you say something to the doctor, you must end it by typing `RET RET`. This lets the doctor know you are finished.

32.3 Reporting Bugs

Sometimes you will encounter a bug in Emacs. Although we cannot promise we can or will fix the bug, and we might not even agree that it is a bug, we want to hear about bugs you encounter in case we do want to fix them.

To make it possible for us to fix a bug, you must report it. In order to do so effectively, you must know when and how to do it.

32.3.1 When Is There a Bug

If Emacs executes an illegal instruction, or dies with an operating system error message that indicates a problem in the program (as opposed to something like "disk full"), then it is certainly a bug.

If Emacs updates the display in a way that does not correspond to what is in the buffer, then it is certainly a bug. If a command seems to do the wrong thing but the problem corrects itself if you type `C-l`, it is a case of incorrect display updating.

Taking forever to complete a command can be a bug, but you must make certain that it was really Emacs's fault. Some commands simply take a long time. Type `C-g` and then `C-h l` to see whether the input Emacs received was what you intended to type; if the input was such that you *know* it should have been processed quickly, report a bug. If you don't know whether the command should take a long time, find out by looking in the manual or by asking for assistance.

If a command you are familiar with causes an Emacs error message in a case where its usual definition ought to be reasonable, it is probably a bug.

If a command does the wrong thing, that is a bug. But be sure you know for certain what it ought to have done. If you aren't familiar with the command, or don't know for certain how the command is supposed to work, then it might actually be working right. Rather than jumping to conclusions, show the problem to someone who knows for certain.

Finally, a command's intended definition may not be best for editing with. This is a very important sort of problem, but it is also a matter of judgment. Also, it is easy to come to such a conclusion out of ignorance of some of the existing features. It is probably best not to complain about such a problem until you have checked the documentation in the usual ways, feel confident that you understand it, and know for certain that what you want is not available. If you are not sure what the command is supposed to do after a careful reading of the manual, check the index and glossary for any terms that may be unclear.

If you still do not understand, that indicates a bug in the manual, which you should report. The manual's job is to make everything clear to people who are not Emacs experts—including you. It is just as important to report documentation bugs as program bugs.

If the on-line documentation string of a function or variable disagrees with the manual, one of them must be wrong; that is a bug.

32.3.2 Understanding Bug Reporting

When you decide that there is a bug, it is important to report it and to report it in a way which is useful. What is most useful is an exact description of what commands you type, starting with the shell command to run Emacs, until the problem happens.

The most important principle in reporting a bug is to report *facts*, not hypotheses or categorizations. It is always easier to report the facts, but people seem to prefer to strain to posit explanations and report them instead. If the explanations are based on guesses about how Emacs is implemented, they will be useless; we will have to try to figure out what the facts must have been to lead to such speculations. Sometimes this is impossible. But in any case, it is unnecessary work for us.

For example, suppose that you type `C-x C-f /glorp/baz.ugh RET`, visiting a file which (you know) happens to be rather large, and Emacs prints out '`I feel pretty today`'. The best way to report the bug is with a sentence like the preceding one, because it gives all the facts and nothing but the facts.

Do not assume that the problem is due to the size of the file and say, "When I visit a large file, Emacs prints out '`I feel pretty today`'." This is what we mean by "guessing explanations". The problem is just as likely to be due to the fact that there is a '`z`' in the file name. If this is so, then when we got your report, we would try out the problem with some "large file", probably with no '`z`' in its name, and not find anything wrong. There is no way in the world that we could guess that we should try visiting a file with a '`z`' in its name.

Alternatively, the problem might be due to the fact that the file starts with exactly 25 spaces. For this reason, you should make sure that you inform us of the exact contents of any file that is needed to reproduce the bug. What if the problem only occurs when you have typed the `C-x C-a` command previously? This is why we ask you to give the exact sequence of characters you typed since starting to use Emacs.

You should not even say "visit a file" instead of `C-x C-f` unless you *know* that it makes no difference which visiting command is used. Similarly, rather than saying "if I have three characters on the line," say "after I type `RET A B C RET C-p`," if that is the way you entered the text.

32.3.3 Checklist for Bug Reports

The best way to send a bug report is to mail it electronically to the Emacs maintainers at '`bug-gnu-emacs@prep.ai.mit.edu`'.

If you'd like to read the bug reports, you can find them on the repeater newsgroup '`gnu.emacs.bugs`'; keep in mind, however, that as a spectator you should not criticize anything about what you see there. The purpose of bug reports is to give information to the Emacs maintainers. Spectators are welcome only as long as they do not interfere with this.

Please do not post bug reports using netnews; mail is more reliable than netnews about reporting your correct address, which we may need in order to ask you for more information.

If you can't send electronic mail, then mail the bug report on paper to this address:

GNU Emacs Bugs
Free Software Foundation
675 Mass Ave
Cambridge, MA 02139

We do not promise to fix the bug; but if the bug is serious, or ugly, or easy to fix, chances are we will want to.

To enable maintainers to investigate a bug, your report should include all these things:

- The version number of Emacs. Without this, we won't know whether there is any point in looking for the bug in the current version of GNU Emacs.

  You can get the version number by typing `M-x emacs-version RET`. If that command does not work, you probably have something other than GNU Emacs, so you will have to report the bug somewhere else.
- The type of machine you are using, and the operating system name and version number.
- The operands you gave to the `configure` command when you installed Emacs.
- A complete list of any modifications you have made to the Emacs source. (We may not have time to investigate the bug unless it happens in an unmodified Emacs. But if you've made modifications and don't tell us, then you are sending us on a wild goose chase.)

Be precise about these changes. A description in English is not enough—send a context diff for them.

Adding files of your own (such as a machine description for a machine we don't support) is a modification of the source.

- Details of any other deviations from the standard procedure for installing GNU Emacs.
- The complete text of any files needed to reproduce the bug.

  If you can tell us a way to cause the problem without visiting any files, please do so. This makes it much easier to debug. If you do need files, make sure you arrange for us to see their exact contents. For example, it can often matter whether there are spaces at the ends of lines, or a newline after the last line in the buffer (nothing ought to care whether the last line is terminated, but try telling the bugs that).
- The precise commands we need to type to reproduce the bug.

  The easy way to record the input to Emacs precisely is to to write a dribble file. To start the file, execute the Lisp expression

  ```
  (open-dribble-file "~/dribble")
  ```

  using `M-ESC` or from the '`*scratch*`' buffer just after starting Emacs. From then on, Emacs copies all your input to the specified dribble file until the Emacs process is killed.
- For possible display bugs, the terminal type (the value of environment variable `TERM`), the complete termcap entry for the terminal from '`/etc/termcap`' (since that file is not identical on all machines), and the output that Emacs actually sent to the terminal.

  The way to collect the terminal output is to execute the Lisp expression

  ```
  (open-termscript "~/termscript")
  ```

  using `M-ESC` or from the '`*scratch*`' buffer just after starting Emacs. From then on, Emacs copies all terminal output to the specified termscript file as well, until the Emacs process is killed. If the problem happens when Emacs starts up, put this expression into your '`.emacs`' file so that the termscript file will be open when Emacs displays the screen for the first time.

  Be warned: it is often difficult, and sometimes impossible, to fix a terminal-dependent bug without access to a terminal of the type that stimulates the bug.
- A description of what behavior you observe that you believe is incorrect. For example, "The Emacs process gets a fatal signal," or, "The resulting text is as follows, which I think is wrong."

  Of course, if the bug is that Emacs gets a fatal signal, then one can't miss it. But if the bug is incorrect text, the maintainer might fail to notice what is wrong. Why leave it to chance?

Even if the problem you experience is a fatal signal, you should still say so explicitly. Suppose something strange is going on, such as, your copy of the source is out of sync, or you have encountered a bug in the C library on your system. (This has happened!) Your copy might crash and the copy here would not. If you *said* to expect a crash, then when Emacs here fails to crash, we would know that the bug was not happening. If you don't say to expect a crash, then we would not know whether the bug was happening. We would not be able to draw any conclusion from our observations.

If the manifestation of the bug is an Emacs error message, it is important to report not just the text of the error message but a backtrace showing how the Lisp program in Emacs arrived at the error. To make the backtrace, execute the Lisp expression `(setq debug-on-error t)` before the error happens (that is to say, you must execute that expression and then make the bug happen). This causes the Lisp debugger to run, showing you a backtrace. Copy the text of the debugger's backtrace into the bug report.

This use of the debugger is possible only if you know how to make the bug happen again. Do note the error message the first time the bug happens, so if you can't make it happen again, you can report at least the error message.

- Check whether any programs you have loaded into the Lisp world, including your '`.emacs`' file, set any variables that may affect the functioning of Emacs. Also, see whether the problem happens in a freshly started Emacs without loading your '`.emacs`' file (start Emacs with the `-q` switch to prevent loading the init file.) If the problem does *not* occur then, you must report the precise contents of any programs that you must load into the Lisp world in order to cause the problem to occur.
- If the problem does depend on an init file or other Lisp programs that are not part of the standard Emacs system, then you should make sure it is not a bug in those programs by complaining to their maintainers first. After they verify that they are using Emacs in a way that is supposed to work, they should report the bug.
- If you wish to mention something in the GNU Emacs source, show the portion in its context. Don't just give a line number.

  The line numbers in the development sources don't match those in your sources. It would take extra work for the maintainers to determine what code is in your version at a given line number, and we could not be certain.
- Additional information from a debugger might enable someone to find a problem on a machine which he does not have available. However, you need to think when you collect this information if you want it to be useful.

For example, many people send just a backtrace, but that is never useful by itself. A simple backtrace with arguments conveys little about what is happening inside GNU Emacs, because most of the arguments listed in the backtrace are pointers to Lisp objects. The numeric values of these pointers have no significance whatever; all that matters is the contents of the objects they point to (and most of the contents are themselves pointers).

To provide useful information, you need to show the values of Lisp objects in Lisp notation. Do this for each variable which is a Lisp object, in several stack frames near the bottom of the stack. Look at the source to see which variables are Lisp objects, because the debugger thinks of them as integers.

To show a variable's value in Lisp syntax, first print its value, then use the GDB command `pr` to print the Lisp object in Lisp syntax. (If you must use another debugger, call the function `debug_print` with the object as an argument.) The `pr` command is defined by the file '`src/.gdbinit`' in the Emacs distribution, and it works only if you are debugging a running process (not with a core dump).

Here are some things that are not necessary:

- A description of the envelope of the bug—this is not necessary for a reproducible bug.

  Often people who encounter a bug spend a lot of time investigating which changes to the input file will make the bug go away and which changes will not affect it.

  This is often time consuming and not very useful, because the way we will find the bug is by running a single example under the debugger with breakpoints, not by pure deduction from a series of examples. You might as well save time by not doing this.

  Of course, if you can find a simpler example to report *instead* of the original one, that is a convenience. Errors in the output will be easier to spot, running under the debugger will take less time, etc.

  However, simplification is not vital; if you don't want to do this, please report the bug with your original test case.

- A patch for the bug.

  A patch for the bug is useful if it is a good one. But don't omit the necessary information, such as the test case, on the assumption that a patch is all we need. We might see problems with your patch and decide to fix the problem another way, or we might not understand it at all.

  And if we can't understand what bug you are trying to fix, or why your patch should be an improvement, we mustn't install it. A test case will help us to understand.

- A guess about what the bug is or what it depends on.

  Such guesses are usually wrong. Even experts can't guess right about such things without first using the debugger to find the facts.

32.3.4 Sending Patches for GNU Emacs

If you would like to write bug fixes or improvements for GNU Emacs, that is very helpful. When you send your changes, please follow these guidelines to make it easy for the maintainers to use them.

If you don't follow these guidelines, your information might still be useful, but using it will take extra work. Maintaining GNU Emacs is a lot of work in the best of circumstances, and we can't keep up unless you do your best to help.

- Send an explanation with your changes of what problem they fix or what improvement they bring about. For a bug fix, just include a copy of the bug report, and explain why the change fixes the bug.

  (Referring to a bug report is not as good as including it, because then we will have to look it up, and we have probably already deleted it if we've already fixed the bug.)
- Always include a proper bug report for the problem you think you have fixed. We need to convince ourselves that the change is right before installing it. Even if it is correct, we might have trouble understanding it if we don't have a way to reproduce the problem.
- Include all the comments that are appropriate to help people reading the source in the future understand why this change was needed.
- Don't mix together changes made for different reasons. Send them *individually*.

  If you make two changes for separate reasons, then we might not want to install them both. We might want to install just one. If you send them all jumbled together in a single set of diffs, we have to do extra work to disentangle them—to figure out which parts of the change serve which purpose. If we don't have time for this, we might have to ignore your changes entirely.

  If you send each change as soon as you have written it, with its own explanation, then the two changes never get tangled up, and we can consider each one properly without any extra work to disentangle them.
- Send each change as soon as that change is finished. Sometimes people think they are helping us by accumulating many changes to send them all together. As explained above, this is absolutely the worst thing you could do.

Since you should send each change separately, you might as well send it right away. That gives us the option of installing it immediately if it is important.

- Use '`diff -c`' to make your diffs. Diffs without context are hard to install reliably. More than that, they are hard to study; we must always study a patch to decide whether we want to install it. Unidiff format is better than contextless diffs, but not as easy to read as '`-c`' format.

  If you have GNU diff, use '`diff -cp`', which shows the name of the function that each change occurs in.

- Write the change log entries for your changes. This is both to save us the extra work of writing them, and to help explain your changes so we can understand them.

  The purpose of the change log is to show people where to find what was changed. So you need to be specific about what functions you changed; in large functions, it's often helpful to indicate where within the function the change was.

  On the other hand, once you have shown people where to find the change, you need not explain its purpose. Thus, if you add a new function, all you need to say about it is that it is new. If you feel that the purpose needs explaining, it probably does—but the explanation will be much more useful if you put it in comments in the code.

  Please read the '`ChangeLog`' file to see what sorts of information to put in, and to learn the style that we use. If you would like your name to appear in the header line showing who made the change, send us the header line.

- When you write the fix, keep in mind that we can't install a change that would break other systems. Please think about what effect your change will have if compiled on another type of system.

  Sometimes people send fixes that *might* be an improvement in general—but it is hard to be sure of this. It's hard to install such changes because we have to study them very carefully. Of course, a good explanation of the reasoning by which you concluded the change was correct can help convince us.

  The safest changes are changes to the configuration files for a particular machine. These are safe because they can't create new bugs on other machines.

  Please help us keep up with the workload by designing the patch in a form that is clearly safe to install.

32.4 How To Get Help with GNU Emacs

If you need help installing, using or changing GNU Emacs, there are two ways to find it:

- Send a message to a suitable network mailing list. First try `bug-gnu-emacs@prep.ai.mit.edu`, and if that brings no response, try `help-gnu-emacs@prep.ai.mit.edu`.
- Look in the service directory for someone who might help you for a fee. The service directory is found in the file named '`etc/SERVICE`' in the Emacs distribution.

Appendix A Command Line Options and Arguments

GNU Emacs supports command line arguments to request various actions when invoking Emacs. These are for compatibility with other editors and for sophisticated activities. We don't recommend using them for ordinary editing.

Arguments that are not options specify files to visit. Emacs visits the specified files while it starts up. (The last file name on your command line is the one you see displayed, but the rest are all there in other buffers.)

You can use options to specify other things, such as the size and position of the Emacs window if you are running it under the X Window System. A few arguments support advanced usage, like running Lisp functions on files in batch mode.

There are two kinds of options: *ordinary options* and *initial options*. Ordinary options can appear in any order and can be intermixed with file names to visit. These and file names are called *ordinary arguments*. Emacs processes all of these in the order they are written. Initial options must come at the beginning of the command line.

A.1 Ordinary Arguments

Here is a table of the ordinary arguments and options:

'*file*' Visit *file* using `find-file`. See Section 15.2 [Visiting], page 94.

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

'`-l` *file*'
'`-load` *file*'
: Load a file *file* of Lisp code with the function `load`. See Section 23.4 [Lisp Libraries], page 205.

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

'`-insert` *file*'
: Insert the contents of *file* into the current buffer. This is like what `M-x insert-buffer` does; See Section 15.10 [Misc File Ops], page 116.

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

A.2 Initial Options

The initial options are recognized only at the beginning of the command line. If you use more than one of them, they must appear in the order that they appear in this table.

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

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

'-nw'
: Don't communicate directly with X, disregarding the `DISPLAY` environment variable even if it is set. '-nw' stands for "non-window."

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

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

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

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

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

The init file can get access to the values of the command line arguments as the elements of a list in the variable `command-line-args`. (The list contains only the arguments from the first table above. Emacs processes the arguments from the second table before building the list.) The init file can override the normal processing of the other arguments by setting this variable.

A.3 Command Argument Example

Here is an example of using Emacs with arguments and options. It assumes you have a Lisp program file called 'hack-c.el' which, when loaded, performs some useful operation on current buffer, expected to be a C program.

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

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

A.4 Resuming Emacs with Arguments

You can specify ordinary arguments for Emacs when you resume it after a suspension. To prepare for this, put the following code in your '.emacs' file (see Section 31.2.3 [Hooks], page 294):

```
(add-hook 'suspend-hook 'resume-suspend-hook)
```

As further preparation, you must execute the shell script 'emacs.csh' (if you use CSH as your shell) or 'emacs.bash' (if you use BASH as your shell). These scripts define an alias named edit, which will resume Emacs giving it new command line arguments such as files to visit.

Only ordinary arguments work properly when you resume Emacs. Initial arguments are not recognized. It's too late to execute them anyway.

Note that resuming Emacs (with or without arguments) must be done from within the shell that is the parent of the Emacs job. This is why edit is an alias rather than a program or a shell script. It is not possible to implement a resumption command that could be run from other subjobs of the shell; no way to define a command that could be made the value of EDITOR, for example. Therefore, this feature does not take the place of the the Emacs Server feature. See Section 30.8 [Emacs Server], page 285.

The aliases use the Emacs Server feature if you appear to have a server Emacs running. However, they cannot determine this with complete accuracy. They may think that a server is still running when in actuality you have killed that Emacs, because the file '/tmp/.esrv. . .' still exists. If this happens, find that file and delete it.

A.5 Specifying the Display Name

The environment variable `DISPLAY` tells all X clients where to display their windows. Its value is set up by default in ordinary circumstances, when you start an X server and run jobs locally. Occasionally you may need to specify the display yourself; for example, if you do a remote login and want to run a client program remotely, displaying on your local screen.

With Emacs, the main reason people change the default display is to let them log into another system, run Emacs on that system, but have the window displayed at their local terminal. You might need to use login to another system because the files you want to edit are there, or because the Emacs executable file you want to run is there.

The syntax of the `DISPLAY` environment variable is:

```
host:display.screen
```

where *host* is the host name of the X Window System server machine, *display* is an arbitrarily-assigned number that distinguishes your server (X terminal) from other servers on the same machine, and *screen* is a rarely-used field that allows an X server to control multiple terminal screens. The period and the *screen* field are optional. If included, *screen* is usually zero.

If your host is named '`glasperle`' and your server is the first (or perhaps the only) server listed in the configuration, your `DISPLAY` is '`glasperle:0.0`'.

You can specify the display name explicitly when you run Emacs, either by changing the `DISPLAY` variable, or with the option '`-d` *display*' or '`-display` *display*'. These are initial options; they must come at the beginning of the command line. See Section A.2 [Initial Options], page 332. Here is an example:

```
emacs -display glasperle:0 &
```

You can inhibit the direct use of X with the '`-nw`' option. This is also an initial option. This option tells Emacs to display using ordinary ASCII on its controlling terminal.

Sometimes, security arrangements prevent a program on a remote system from displaying on your local system. In this case, trying to run Emacs produces messages like:

```
Xlib:  connection to "glasperle:0.0" refused by server
```

You might be able to overcome this problem by using the `xhost` command on the local system to give permission for access from your remote machine.

A.6 Font Specification Options

By default, Emacs displays text in the font named '`9x15`', which makes each character nine pixels wide and fifteen pixels high. You can specify a different font on your command line through the option '-fn *name*'. The '-font' option is a synonym for '-fn'.

Here is how to specify the font '6x13':

```
emacs -fn 6x13 &
```

You can also do this in your '.Xdefaults' file:

```
emacs.font:      6x13
```

Use only fixed width fonts—that is, fonts in which all characters have the same width. Emacs cannot yet handle display properly for variable width fonts. Fixed width fonts include the one named 'fixed', and fonts with names in the form *n*x*n*, such as '6x13', '8x13', and '9x15'. Under the font-naming conventions in X11 Release 4 or later, any font with 'm' or 'c' in the eleventh field of the name is a fixed width font.

Here's how to use the xlsfonts program to list all the fixed width fonts available on your system:

```
xlsfonts -fn '*x*'
xlsfonts -fn '*-*-*-*-*-*-*-*-*-*-*-m*'
xlsfonts -fn '*-*-*-*-*-*-*-*-*-*-*-c*'
```

To see what a particular font looks like, use the xfd command. For example:

```
xfd -fn 6x13
```

displays the entire font '6x13'.

While running Emacs, you can set the font of the current frame (see Section 18.4 [Frame Parameters], page 132) or for a specific kind of text (see Section 18.7 [Faces], page 134).

A.7 Window Color Options

On a color display, you can specify which color to use for various parts of the Emacs display. To find out what colors are available on your system, look at the '/usr/lib/X11/rgb.txt' file. If you do not specify colors, the default for the background is white and the default for all other colors is black.

On a monochrome (black and white) display, the foreground is black, the background is white, and the border is grey. You can reverse the foreground and background colors through the '-r' option or the 'reverseVideo' resource.

Here is a list of the options for specifying colors:

'-fg *color*'
: Specify the foreground color.

'-bg *color*'
: Specify the background color.

'`-bd` *color*'
: Specify the color of the border of the X window.

'`-cr` *color*'
: Specify the color of the Emacs cursor which indicates where point is.

'`-ms` *color*'
: Specify the color for the mouse cursor when the mouse is in the Emacs window.

For example, to use a coral mouse cursor and a slate blue text cursor, enter:

```
emacs -ms coral -cr 'slate blue' &
```

A.8 Options for Window Geometry

The '`-geometry`' option controls the size and position of the initial Emacs frame. Here is the format for specifying the window geometry:

width`x`*height*`{+-}`*xoffset*`{+-}`*yoffset*

where *width* specifies the number of characters displayed on a line, *height* specifies the number of lines displayed, a positive *xoffset* specifies the distance from the left side of the screen, a negative *xoffset* specifies the distance from the right side of the screen, a positive *yoffset* specifies the distance from the top of the screen, and a negative *yoffset* specifies the distance from the bottom of the screen.

Emacs uses the same units as `xterm` does to interpret the geometry. The *width* and *height* are measured in characters, so a large font creates a larger frame than a small font. The *xoffset* and *yoffset* are measured in pixels.

Since the mode line and the echo area occupy the last 2 lines of the frame, the height of the initial text window is 2 less than the height specified in your geometry.

You do not have to specify all of the fields in the geometry specification. The default width for Emacs is 80 characters and the default height is 24 characters. You can omit either the width or the height or both.

If you omit both *xoffset* nor *yoffset*, the window manager decides where to put the Emacs frame, possibly by letting you place it with the mouse. For example, '`164x55`' specifies a window 164 columns wide, enough for two ordinary width windows side by side, and 55 lines tall.

If you start the geometry with an integer, Emacs interprets it as the width. If you start with an '`x`' followed by an integer, Emacs interprets it as the height. Thus, '`81`' specifies just the width; '`x45`' specifies just the height.

If you start with '`+`' or '`-`', that introduces an offset, which means both sizes are omitted. Thus, '`-3`' specifies the *xoffset* only. (If you give just one offset, it is always *xoffset*.) '`+3-3`' specifies both the *xoffset* and the *yoffset*, placing the frame near the bottom left of the screen.

You can specify a default for any or all of the fields in '`.Xdefaults`' file, and then override selected fields through a '`-geometry`' option.

A.9 Internal and External Borders

An Emacs frame has an internal border and an external border. The internal border is an extra strip of the background color around all four edges of the frame. Emacs itself adds the internal border. The external border is added by the window manager outside the internal border; it may contain various boxes you can click on to move or iconify the window.

When you specify the size of the frame, that does not count the borders. The frame's position is measured from the outside edge of the external border.

Use the '`-ib` *n*' option to specify an internal border *n* pixels wide. The default is 1. Use '`-bw` *n*' to specify the width of the external border (though the window manager may not pay attention to what you specify). The default width of the external border is 2.

A.10 Icons

Most window managers allow the user to "iconify" a frame, removing it from sight, and leaving a small, distinctive "icon" window in its place. Clicking on the icon window will make the original frame visible again. If a user has many clients running at once, they can avoid cluttering up their screen by iconifying all but the clients currently in use.

The '`-i`' and '`-itype`' option tells Emacs to use an icon window containing a picture of the GNU gnu. If omitted, Emacs lets the window manager choose what sort of icon to use — usually just a small rectangle containing the frame's title.

The '`-iconic`' option tells Emacs to begin running as an icon, rather than opening a frame right away. In this situation, the icon window provides only indication that Emacs has started; the usual text frame doesn't appear until you de-iconify it.

A.11 X Resources

Programs running under the X Window System organize their user options under a hierarchy of classes and resources. You can specify default values for these options in your X resources file, usually named '`~/.Xdefaults`'.

Each line in the file specifies a value for one option or for a collection of related options, for one program or for several programs (perhaps even all programs).

Programs define named resources with particular meanings. They also define how to group resources into named classes. For instance, in Emacs, the '`internalBorder`' resource controls the width of the internal border, and the '`borderWidth`' resource controls the width of the external border. Both of these resources are part of the '`BorderWidth`' class. Case distinctions are significant in these names.

In '`~/.Xdefaults`', you can specify a value for a single resource on one line, like this:

```
emacs.borderWidth: 2
```

Or you can use a class name to specify the same value for all resources in that class. Here's an example:

```
emacs.BorderWidth: 2
```

If you specify a value for a class, it becomes the default for all resources in that class. You can specify values for individual resources as well; these override the class value, for those particular resources. Thus, this example specifies 2 as the default width for all borders, but overrides this value with 4 for the external border:

```
emacs.Borderwidth: 2
emacs.borderwidth: 4
```

The order in which the lines appear in the file does not matter. Also, command-line options always override the X resources file.

The string '`emacs`' in the examples above is also a resource name. It actually represents the name of the executable file that you invoke to run Emacs. If Emacs is installed under a different name, it look for resources under that name instead of '`emacs`'.

You can tell Emacs to use a different name instead of the name of the executable file, with the option '`-name` *name*'. Then that Emacs job uses *name* instead of '`Emacs`' to look up all of its option values in the X resource file. This option also specifies the title for the initial frame.

The resources that name Emacs invocations also belong to a class; its name is '`Emacs`'. To specify options for all Emacs jobs, no matter what name is used to run them, write '`Emacs`' instead of '`emacs`', like this:

```
Emacs.BorderWidth: 2
Emacs.borderWidth: 4
```

You can specify a string of additional resource values for Emacs to use with the command line option '`-xrm` *data*'. The text *data* should have the same format that you would use inside a file of X resources. Resources specified this way take precedence over all others.

The following table lists the resource names that designate options for Emacs, each with the class that it belongs to:

`background` (class `Background`)
: Background color name.

`bitmapIcon` (class `BitmapIcon`)
: Use a bitmap icon (a picture of a GNU) if '`on`', let the window manager choose an icon if '`off`'.

`borderColor` (class `BorderColor`)
: Color name for external border.

`borderWidth` (class `BorderWidth`)
: Width in pixels of external border.

`cursorColor` (class `Foreground`)
: Color name for text cursor (point).

`font` (class `Font`)
: Font name for text.

`foreground` (class `Foreground`)
: Color name for text.

`geometry` (class `Geometry`)
: Window size and position.

`iconName` (class `Title`)
: Name to display in icon.

`internalBorder` (class `BorderWidth`)
: Width in pixels of internal border.

`paneFont` (class `Font`)
: Font name for menu pane titles.

`pointerColor` (class `Foreground`)
: Color of mouse cursor.

`reverseVideo` (class `ReverseVideo`)
: Switch foreground and background default colors if '`on`', use colors as specified if '`off`'.

`selectionFont` (class `Font`)
: Font name for menu items.

`title` (class `Title`)
: Name to display in title bar of initial Emacs frame.

Appendix B Emacs 18 Antinews

For those users who live backwards in time, here is information about downgrading to Emacs version 18. We hope you will enjoy the greater simplicity that results from the absence of many Emacs 19 features.

Packages Removed

To reduce the size of the distribution, we have eliminated numerous packages including GNUS, VC (version control), Hexl (for editing binary files), Edebug, Emerge, Mpuz, Spook, and Gomoku.

Major modes removed in Emacs 18 include C`++` mode, Awk mode, Icon mode, Asm mode, Makefile mode, Perl mode and SGML mode.

The function `enable-flow-control` does not exist; see the file '`PROBLEMS`' in the Emacs distribution for directions for coping with flow control.

The Calendar feature provided is a very simple one. All it can do is display three months, by default centered around the current month. If you give it a numeric argument, that specifies the number of months forward or back.

Fundamental Changes

Auto save and garbage collection happen only while you are typing, never while you are idle. This is to make them more like affectionate pets. Think of them as cats that like to sit on your terminal only when you are working there.

Transient Mark mode and Line Number mode are absent in Emacs 18. If you are an Emacs user, you are smart enough to keep track of the mark in your head, and you don't need line numbers because you can search for precisely the text you want.

There are no menu bars or scroll bars; no faces, text properties or overlays.

There are no minibuffer history commands.

There is only one frame, so the Emacs 19 `C-x 5` command series is meaningless. Instead, `C-x 5` in Emacs 18 splits the selected window horizontally (like `C-x 3` in Emacs 19).

Another simplification in Emacs 18 is that all input events are characters. Function keys and arrow keys are represented as sequences of characters; the terminal-specific Emacs Lisp file for your terminal is responsible for defining them. Mouse buttons are defined by a special keymap, `mouse-map`. See the file '`x-mouse.el`' for how to bind mouse clicks.

Character codes 128 and above always display using '*nnn*' notation. For codes 0 through 31, you can choose between '*nnn*' and '^*c*' by setting the variable `ctl-arrow`; but that is the only thing you can specify about how character codes should display.

You can't refer to files on other machines using special "magic" file names. Instead, you must use the `ftp` library with commands such as `M-x ftp-find-file` and `M-x ftp-write-file`.

The character for terminating an incremental search is now ESC, not RET as in Emacs 19. If you type RET, that searches for a newline; thus, you can insert a newline in the search string just as you would insert it in the text.

Key Binding Changes

The key for `backward-paragraph` is now `M-[`. The key for `forward-paragraph` is now `M-]`.

The command `repeat-complex-command` is now on `C-x ESC`.

The register commands have different key bindings:

`C-x /` `point-to-register`

`C-x j` `jump-to-register`

`C-x x` `copy-to-register`

`C-x g` `insert-register`

`C-x r` `copy-rectangle-to-register`

The narrowing commands have also been moved:

`C-x n` `narrow-to-region`

`C-x p` `narrow-to-page`

`C-x w` `widen`

And the abbrev commands as well:

`C-x C-a` `add-mode-abbrev`

`C-x +` `add-global-abbrev`

`C-x C-h` `inverse-add-mode-abbrev`

`C-x -` `inverse-add-global-abbrev`

`` C-x ` `` `expand-abbrev`

There are no key bindings for the rectangle commands.

`C-x a` now runs the command `append-to-buffer`.

The key bindings `C-x 4 r` and `C-x 4 C-o` do not exist.

The help commands `C-h C-f`, `C-h C-k` and `C-h p` do not exist in Emacs 18.

The command `C-M-l` (`reposition-window`) is absent. Likewise `C-M-r` (`isearch-backward-regexp`).

The "two column" commands starting with `C-x 6` don't exist in Emacs 18.

The TeX mode bindings of `C-c {` and `C-c }` have been moved to `M-{` and `M-}`. (These commands are `up-list` and `tex-insert-braces`; they are the TeX equivalents of `M-(` and `M-)`.)

Incremental Search Changes

As mentioned above, the character for terminating an incremental search is now `ESC`, not `RET` as in Emacs 19. If you type `RET`, that searches for a newline; thus, you can insert a newline in the search string just as you would insert it in the text.

There is no ring of previous search strings in Emacs 18. You can reuse the most recent search string, but that's all.

If `case-fold-search` is non-`nil`, then incremental search is *always* case-insensitive. Typing an upper-case letter in the search string has no effect on this.

Spaces in the incremental search string match only spaces.

The meanings of the special search characters are no longer controlled by a keymap. Instead, particular variables named `search-...-char` specify the character that should have a particular function. For example, `C-s` repeats the search because the value of `search-repeat-char` is `?\C-s`.

Editing Command Changes

`C-n` (`next-line`) does not check the variable `next-line-add-newlines`.

The sexp commands such as `C-M-f` no longer know anything about comments, in modes such as Lisp mode where the end of a comment is the end of the line. They treat the text inside a comment as if it were actual code. If comments containing unbalanced parentheses cause trouble, you can use the commands `C-M-n` and `C-M-p`, which do ignore comments.

You can't store file names in registers, and there are no frame configurations at all. The command `M-x string-rectangle` does not exist either.

The undo command in Emacs 18 is not careful about where to leave point when you undo a deletion. It ends up at one end or the other of the text just undeleted. You must be on the lookout for this, and move point appropriately.

Kill commands do nothing useful in read-only buffers. They just beep.

`M-z` *c* in Emacs 18 kills up to but not including the first occurrence of *c*. If *c* does not occur in the buffer after point, `M-z` kills the whole rest of the buffer.

The function `erase-buffer` is not a command in Emacs 18. You can call it from a Lisp program, but not interactively. The motivation for this is to protect you from accidentally deleting (not killing) the entire text of a buffer that you want to keep. With subsequent changes in even earlier Emacs versions (such as version 18.54), you might be unable to undo the `erase-buffer`.

`M-x fill-nonuniform-paragraphs` and Adaptive Fill mode do not exist.

Other Brief Notes

Outline mode exists only as a major mode, not as a minor mode.

`M-!` (`shell-command`) always runs the command synchronously, even if the command ends with '`&`'.

Emacs 18 has no special mode for change log files. It is a good idea to use Indented Text mode, and specify 8 as the value of the variable `left-margin`.

The command `M-x comment-region` does not exist. The command `M-x super-apropos` does not exist.

`C-x q` (`kbd-macro-query`) now uses `C-d` to terminate all iterations of the keyboard macro, rather than `ESC`.

The `M-x setenv` command is missing in Emacs 18.

`M-$` now uses the Unix spell program instead of the GNU program Ispell. If the word around point is a misspelling, it asks you for a replacement.

To check spelling of larger units of text, use `M-x spell-region` or `M-x spell-buffer`. These commands check all words in the specified piece of text. For each word that is not correct, they ask you to specify a replacement, and then replace each occurrence.

`M-x gdb` still exists in Emacs 18. `M-x dbx` exists, but is somewhat different (use `C-h m` to find the details). `M-x sdb` does not exist at all, but who wants to use SDB?

In Buffer Menu mode, the commands `%` and `C-o` don't work in Emacs 18. The `v` command has been eliminated and merged with the `q` command, which now exits the buffer menu, displaying all the buffers that you have marked.

The View commands (such as `M-x view-buffer` and `M-x view-file`) now use recursive edits. When you exit viewing, the recursive edit returns to its caller.

Emacs 18, like most programs, interprets command line options only when it is started–not later on.

The variable to control whether files can set local variables is called `inhibit-local-variables`. A non-`nil` value means ask the user before obeying any local variables lists.

The user option for controlling use of the `eval` local variable is now called `inhibit-local-eval`. A non-`nil` value means to ask the user before obeying any `eval` local variable.

File Handling Changes

As mentioned above, you can't refer to files on other machines using special "magic" file names. Instead, you must use the `ftp` library with commands such as `M-x ftp-find-file` and `M-x ftp-write-file`.

When you run `M-x revert-buffer` with no prefix argument, if the buffer has an auto save file more recent that the visited file, `revert-buffer` asks whether to revert from the auto save file instead.

When `C-x s` (`save-some-buffers`) offers to save each buffer, you have only two choices: save it, or don't save it.

`M-x recover-file` turns off Auto Save mode in the current buffer. To turn it on again, use `M-x auto-save-mode`.

The command `M-x rename-uniquely` does not exist; instead, use `M-x rename-buffer` and try various names until you find one that isn't in use. Completion can make this easier.

The directory name abbreviation feature is gone in Emacs 18.

Emacs 18 has no idea of file truenames, and does not try to detect when you visit a file via a symbolic link. You should check manually when you visit a file, so as to edit it in the directory where it is actually stored. This way you can make sure that backup files and change log entries go in the proper directory.

`M-x compare-windows` ignores any prefix argument and always considers case and whitespace differences significant. As for the other ways of comparing files, `M-x diff` and `M-x diff-backup`, they don't exist at all.

Mail Changes

'`%`' is now a word-component character in Mail mode. This is to be compatible with Text mode.

The variable `mail-signature` is not meaningful; if you wish to insert your signature in a mail message, you must type `C-c C-w`.

Mail aliases expand only when you send the message—never when you type them in.

Rmail now gets new mail into your primary mail file from '~/mbox' as well as from your system inbox file. This is handy if you occasionally check your newest mail with the `mail` program; whatever you have looked at and saved with `mail` will be brought into Rmail the next time you run Rmail.

The Rmail summary buffer is now much simpler. Only a few special commands are available there: `n`, `p`, and `j` for motion, `d` and `u` for deletion, and `SPC` and `DEL` for scrolling the message. To do anything else, you must go to the Rmail buffer. Also, changes in the Rmail buffer don't update the summary; to do that, you must make a new summary.

The Rmail command `rmail-resend` (accessible via `f` with a prefix argument in Emacs 19) does not exist in Emacs 18. Neither does `rmail-retry-failure` (`M-m` in Emacs 19).

The `e` command is now "expunge", just like `x`. To edit the current message, type `w`, which works in Emacs 19 as well. If you type `e` meaning to edit, and it expunges instead—well, you shouldn't have deleted those messages if you still wanted them.

The `<` and `b` commands have been removed in Emacs 18. Likewise `C-M-t` (`rmail-summarize-by-topic`) and `M-x unrmail`. Rmail in Emacs 18 is so good, that we can't imagine anyone who has tried it would ever wish to use another mail reader.

The default output file for `o` is now always the last file that you used with `o`. The variable `rmail-output-file-alist` has no special meaning.

Emacs 18 Rmail does not know anything about Content Length fields in messages.

C Mode Changes

In C mode, the keys `M-a` and `M-e` now have their usual meanings: motion by sentences. This is useful while editing the comments in a C program, but not useful for editing code. We hope this will encourage you to write lots of comments.

The commands `M-x c-up-conditional` and `M-x c-backslash-region` have been removed entirely in Emacs 18.

Compilation Changes

`M-x compile` now has a much simpler and faster parser for error messages. However, it understands fewer different formats for error messages, and is not as easy to customize.

There is no special mode for compilation buffers. When you select the compilation buffer itself, it is just ordinary text.

Speaking of selecting the compilation buffer, you do need to do that from time to time to see whether the compilation has finished, because Emacs 18 does not display 'Compiling' in the mode line to tell you the compilation is still going.

Shell Mode

Shell mode in Emacs 18 does nothing special for the keys TAB, M-?, C-a, C-d. The commands M-x dirs and M-x send-invisible are also gone.

The history commands M-p and so on are not available either; instead, use C-c C-y (copy-last-shell-input). This copies the previous bunch of shell input, and inserts it into the buffer before point. No final newline is inserted, and the input copied is not resubmitted until you type RET.

Use C-c C-d to send an "end of file" to the shell process.

Dired Changes

For simplicity, Dired in Emacs 18 supports just one kind of mark: the deletion flag, '*'. The Emacs 19 Dired commands for flagging files do work in Emacs 18, but all the other mark-related commands do not.

The Dired subdirectory commands don't exist in Emacs 18. A Dired buffer can contain only one directory. In particular, this means that the variable dired-listing-switches must not contain the 'R' option. (The 'F' option is also not allowed.)

The commands for using find with Dired have been removed for simplicity, also.

Emacs 18 Dired provides the following commands for manipulating files immediately, and no others. All of these commands apply to the file listed on the current line.

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

f
: Visits the file described on the current line. It is just like typing C-x C-f and supplying that file name. If the file on this line is a subdirectory, f actually causes Dired to be invoked on that subdirectory.

G
: Change the group of the file described on the current line.

M
: Change the file mode of the file described on the current line.

`o` Like `f`, but uses another window to display the file's buffer. The Dired buffer remains visible in the first window. This is like using `C-x 4 C-f` to visit the file.

`O` Change the owner of the file described on the current line. (On most systems, you must be a superuser to do this.)

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

`v` Views the file described on this line using `M-x view-file`. Viewing a file is like visiting it, but is slanted toward moving around in the file conveniently and does not allow changing the file.

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 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.

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.[1] 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

[1] 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.

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.

Why I Must Write GNU

I consider that the golden rule requires that if I like a program I must share it with other people who like it. Software sellers want to divide the users and conquer them, making each user agree not to share with others. I refuse to break solidarity with other users in this way. I cannot in good conscience sign a nondisclosure agreement or a software license agreement. For years I worked within the Artificial Intelligence Lab to resist such tendencies and other inhospitalities, but eventually they had gone too far: I could not remain in an institution where such things are done for me against my will.

So that I can continue to use computers without dishonor, I have decided to put together a sufficient body of free software so that I will be able to get along without any software that is not free. I have resigned from the AI lab to deny MIT any legal excuse to prevent me from giving GNU away.

Why GNU Will Be Compatible with Unix

Unix is not my ideal system, but it is not too bad. The essential features of Unix seem to be good ones, and I think I can fill in what Unix lacks without spoiling them. And a system compatible with Unix would be convenient for many other people to adopt.

How GNU Will Be Available

GNU is not in the public domain. Everyone will be permitted to modify and redistribute GNU, but no distributor will be allowed to restrict its further redistribution. That is to say, proprietary modifications will not be allowed. I want to make sure that all versions of GNU remain free.

Why Many Other Programmers Want to Help

I have found many other programmers who are excited about GNU and want to help.

Many programmers are unhappy about the commercialization of system software. It may enable them to make more money, but it requires them to feel in conflict with other programmers in general rather than feel as comrades. The fundamental act of friendship among programmers is the sharing of programs; marketing arrangements now typically used essentially forbid programmers to treat others as friends. The purchaser of software must choose between friendship and obeying the law. Naturally, many decide that friendship is more important. But those who believe in law often do not feel at ease with either choice. They become cynical and think that programming is just a way of making money.

By working on and using GNU rather than proprietary programs, we can be hospitable to everyone and obey the law. In addition, GNU serves as an example to inspire and a banner to rally others to join us in sharing. This can give us a feeling of harmony which is impossible if we use software that is not free. For about half the programmers I talk to, this is an important happiness that money cannot replace.

How You Can Contribute

I am asking computer manufacturers for donations of machines and money. I'm asking individuals for donations of programs and work.

One consequence you can expect if you donate machines is that GNU will run on them at an early date. The machines should be complete, ready to use systems, approved for use in a residential area, and not in need of sophisticated cooling or power.

I have found very many programmers eager to contribute part-time work for GNU. For most projects, such part-time distributed work would be very hard to coordinate; the independently-written parts would not work together. But for the particular task of replacing Unix, this problem is absent. A complete Unix system contains hundreds of utility programs, each of which is documented separately. Most interface specifications are fixed by Unix compatibility. If each contributor can write a compatible replacement for a single Unix utility, and make it work properly in place of the original on a Unix system, then these utilities will work right when put together. Even allowing for Murphy to create a few unexpected problems, assembling these components will be a feasible task. (The kernel will require closer communication and will be worked on by a small, tight group.)

If I get donations of money, I may be able to hire a few people full or part time. The salary won't be high by programmers' standards, but I'm looking for people for whom building community spirit is as important as making money. I view this as a way of enabling dedicated people to devote their full energies to working on GNU by sparing them the need to make a living in another way.

Why All Computer Users Will Benefit

Once GNU is written, everyone will be able to obtain good system software free, just like air.[2]

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

---

[2] 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.

metered gas mask all day and all night is intolerable even if everyone can afford to pay the air bill. And the TV cameras everywhere to see if you ever take the mask off are outrageous. It's better to support the air plant with a head tax and chuck the masks.

Copying all or parts of a program is as natural to a programmer as breathing, and as productive. It ought to be as free.

Some Easily Rebutted Objections to GNU's Goals

> "Nobody will use it if it is free, because that means they can't rely on any support."
>
> "You have to charge for the program to pay for providing the support."

If people would rather pay for GNU plus service than get GNU free without service, a company to provide just service to people who have obtained GNU free ought to be profitable.[3]

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."

[3] Several such companies now exist.

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?[4]

> "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.[5]

> "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.

---

[4] 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 the 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?

[5] A group of computer companies recently pooled funds to support maintenance of the GNU C Compiler.

Extracting money from users of a program by restricting their use of it is destructive because the restrictions reduce the amount and the ways that the program can be used. This reduces the amount of wealth that humanity derives from the program. When there is a deliberate choice to restrict, the harmful consequences are deliberate destruction.

The reason a good citizen does not use such destructive means to become wealthier is that, if everyone did so, we would all become poorer from the mutual destructiveness. This is Kantian ethics; or, the Golden Rule. Since I do not like the consequences that result if everyone hoards information, I am required to consider it wrong for one to do so. Specifically, the desire to be rewarded for one's creativity does not justify depriving the world in general of all or part of that creativity.

"Won't programmers starve?"

I could answer that nobody is forced to be a programmer. Most of us cannot manage to get any money for standing on the street and making faces. But we are not, as a result, condemned to spend our lives standing on the street making faces, and starving. We do something else.

But that is the wrong answer because it accepts the questioner's implicit assumption: that without ownership of software, programmers cannot possibly be paid a cent. Supposedly it is all or nothing.

The real reason programmers will not starve is that it will still be possible for them to get paid for programming; just not paid as much as now.

Restricting copying is not the only basis for business in software. It is the most common basis because it brings in the most money. If it were prohibited, or rejected by the customer, software business would move to other bases of organization which are now used less often. There are always numerous ways to organize any kind of business.

Probably programming will not be as lucrative on the new basis as it is now. But that is not an argument against the change. It is not considered an injustice that sales clerks make the salaries that they now do. If programmers made the same, that would not be an injustice either. (In practice they would still make considerably more than that.)

"Don't people have a right to control how their creativity is used?"

"Control over the use of one's ideas" really constitutes control over other people's lives; and it is usually used to make their lives more difficult.

People who have studied the issue of intellectual property rights carefully (such as lawyers) say that there is no intrinsic right to intellectual property. The kinds of supposed intellectual property rights that the government recognizes were created by specific acts of legislation for specific purposes.

For example, the patent system was established to encourage inventors to disclose the details of their inventions. Its purpose was to help society rather than to help inventors. At the time, the life span of 17 years for a patent was short compared with the rate of advance of the state of the art. Since patents are an issue only among manufacturers, for whom the cost and effort of a license agreement are small compared with setting up production, the patents often do not do much harm. They do not obstruct most individuals who use patented products.

The idea of copyright did not exist in ancient times, when authors frequently copied other authors at length in works of non-fiction. This practice was useful, and is the only way many authors' works have survived even in part. The copyright system was created expressly for the purpose of encouraging authorship. In the domain for which it was invented—books, which could be copied economically only on a printing press—it did little harm, and did not obstruct most of the individuals who read the books.

All intellectual property rights are just licenses granted by society because it was thought, rightly or wrongly, that society as a whole would benefit by granting them. But in any particular situation, we have to ask: are we really better off granting such license? What kind of act are we licensing a person to do?

The case of programs today is very different from that of books a hundred years ago. The fact that the easiest way to copy a program is from one neighbor to another, the fact that a program has both source code and object code which are distinct, and the fact that a program is used rather than read and enjoyed, combine to create a situation in which a person who enforces a copyright is harming society as a whole both materially and spiritually; in which a person should not do so regardless of whether the law enables him to.

> "Competition makes things get done better."

The paradigm of competition is a race: by rewarding the winner, we encourage everyone to run faster. When capitalism really works this way, it does a good job; but its defenders are wrong in assuming it always works this way. If the runners forget why the reward is offered and become intent on winning, no matter how, they may find other strategies—such as, attacking other runners. If the runners get into a fist fight, they will all finish late.

Proprietary and secret software is the moral equivalent of runners in a fist fight. Sad to say, the only referee we've got does not seem to object to fights; he just regulates them ("For every ten yards you run, you can fire one shot"). He really ought to break them up, and penalize runners for even trying to fight.

> "Won't everyone stop programming without a monetary incentive?"

Actually, many people will program with absolutely no monetary incentive. Programming has an irresistible fascination for some people, usually the people who are best at it. There is no shortage of professional musicians who keep at it even though they have no hope of making a living that way.

But really this question, though commonly asked, is not appropriate to the situation. Pay for programmers will not disappear, only become less. So the right question is, will anyone program with a reduced monetary incentive? My experience shows that they will.

For more than ten years, many of the world's best programmers worked at the Artificial Intelligence Lab for far less money than they could have had anywhere else. They got many kinds of non-monetary rewards: fame and appreciation, for example. And creativity is also fun, a reward in itself.

Then most of them left when offered a chance to do the same interesting work for a lot of money.

What the facts show is that people will program for reasons other than riches; but if given a chance to make a lot of money as well, they will come to expect and demand it. Low-paying organizations do poorly in competition with high-paying ones, but they do not have to do badly if the high-paying ones are banned.

> "We need the programmers desperately. If they demand that we stop helping our neighbors, we have to obey."

You're never so desperate that you have to obey this sort of demand. Remember: millions for defense, but not a cent for tribute!

> "Programmers need to make a living somehow."

In the short run, this is true. However, there are plenty of ways that programmers could make a living without selling the right to use a program. This way is customary now because it brings programmers and businessmen the most money, not because it is the only way to make a living. It is easy to find other ways if you want to find them. Here are a number of examples.

A manufacturer introducing a new computer will pay for the porting of operating systems onto the new hardware.

The sale of teaching, hand-holding and maintenance services could also employ programmers.

People with new ideas could distribute programs as freeware, asking for donations from satisfied users, or selling hand-holding services. I have met people who are already working this way successfully.

Users with related needs can form users' groups, and pay dues. A group would contract with programming companies to write programs that the group's members would like to use.

All sorts of development can be funded with a Software Tax:

> Suppose everyone who buys a computer has to pay x percent of the price as a software tax. The government gives this to an agency like the NSF to spend on software development.
>
> But if the computer buyer makes a donation to software development himself, he can take a credit against the tax. He can donate to the project of his own choosing—often, chosen because he hopes to use the results when it is done. He can take a credit for any amount of donation up to the total tax he had to pay.
>
> The total tax rate could be decided by a vote of the payers of the tax, weighted according to the amount they will be taxed on.
>
> The consequences:
>
> - The computer-using community supports software development.
> - This community decides what level of support is needed.
> - Users who care which projects their share is spent on can choose this for themselves.

In the long run, making programs free is a step toward the post-scarcity world, where nobody will have to work very hard just to make a living. People will be free to devote themselves to activities that are fun, such as programming, after spending the necessary ten hours a week on required tasks such as legislation, family counseling, robot repair and asteroid prospecting. There will be no need to be able to make a living from programming.

We have already greatly reduced the amount of work that the whole society must do for its actual productivity, but only a little of this has translated itself into leisure for workers because much nonproductive activity is required to accompany productive activity. The main causes of this are bureaucracy and isometric struggles against competition. Free software will greatly reduce these drains in the area of software production. We must do this, in order for technical gains in productivity to translate into less work for us.

Glossary

Abbrev
: An abbrev is a text string which expands into a different text string when present in the buffer. For example, you might define a short word as an abbrev for a long phrase that you want to insert frequently. See Chapter 24 [Abbrevs], page 211.

Aborting
: Aborting means getting out of a recursive edit (q.v.). The commands `C-]` and `M-x top-level` are used for this. See Section 32.1 [Quitting], page 317.

Alt
: Alt is the name of a modifier bit which a keyboard input character may have. To make a character Alt, type it while holding down the `ALT` key. Such characters are given names that start with `Alt-` (usually written `A-` for short). See Section 2.1 [User Input], page 19.

Auto Fill Mode
: Auto Fill mode is a minor mode in which text that you insert is automatically broken into lines of fixed width. See Section 21.5 [Filling], page 147.

Auto Saving
: Auto saving is the practice of saving the contents of an Emacs buffer in a specially-named file, so that the information will not be lost if the buffer is lost due to a system error or user error. See Section 15.5 [Auto Save], page 103.

Backup File
: A backup file records the contents that a file had before the current editing session. Emacs makes backup files automatically to help you track down or cancel changes you later regret making. See Section 15.3.1 [Backup], page 99.

Balance Parentheses
: Emacs can balance parentheses manually or automatically. Manual balancing is done by the commands to move over balanced expressions (see Section 22.2 [Lists], page 164). Automatic balancing is done by blinking the parenthesis that matches one just inserted (see Section 22.6 [Matching Parens], page 173).

Bind
: To bind a key sequence means to give it a binding (q.v.). See Section 31.4.5 [Rebinding], page 305.

Binding
: A key sequence gets its meaning in Emacs by having a binding, which is a command (q.v.), a Lisp function that is run when the user types that sequence. See Section 2.3 [Commands], page 21. Customization often involves rebinding a character to a different command function. The bindings of all key sequences are

recorded in the keymaps (q.v.). See Section 31.4.1 [Keymaps], page 302.

Blank Lines
: Blank lines are lines that contain only whitespace. Emacs has several commands for operating on the blank lines in the buffer.

Buffer
: The buffer is the basic editing unit; one buffer corresponds to one piece of text being edited. You can have several buffers, but at any time you are editing only one, the 'selected' buffer, though several can be visible when you are using multiple windows. See Chapter 16 [Buffers], page 119.

Buffer Selection History
: Emacs keeps a buffer selection history which records how recently each Emacs buffer has been selected. This is used for choosing a buffer to select. See Chapter 16 [Buffers], page 119.

Button Down Event
: A button down event is the kind of input event generated right away when you press a mouse button. See Section 31.4.8 [Mouse Buttons], page 308.

C-
: '`C`' in the name of a character is an abbreviation for Control. See Section 2.1 [User Input], page 19.

C-M-
: '`C-M-`' in the name of a character is an abbreviation for Control-Meta. See Section 2.1 [User Input], page 19.

Case Conversion
: Case conversion means changing text from upper case to lower case or vice versa. See Section 21.6 [Case], page 151, for the commands for case conversion.

Characters
: Characters form the contents of an Emacs buffer; see Section 2.4 [Text Characters], page 22. Also, key sequences (q.v.) are usually made up of characters (though they may include other input events as well). See Section 2.1 [User Input], page 19.

Click Event
: A click event is the kind of input event generated when you press a mouse button and let it go without moving the mouse. See Section 31.4.8 [Mouse Buttons], page 308.

Command
: A command is a Lisp function specially defined to be able to serve as a key binding in Emacs. When you type a key sequence (q.v.), its binding (q.v.) is looked up in the relevant keymaps (q.v.) to find the command to run. See Section 2.3 [Commands], page 21.

Command Name

A command name is the name of a Lisp symbol which is a command (see Section 2.3 [Commands], page 21). You can invoke any command by its name using `M-x` (see Chapter 7 [M-x], page 43).

Comments

A comment is text in a program which is intended only for humans reading the program, and which is marked specially so that it will be ignored when the program is loaded or compiled. Emacs offers special commands for creating, aligning and killing comments. See Section 22.7.2 [Comments], page 175.

Compilation

Compilation is the process of creating an executable program from source code. Emacs has commands for compiling files of Emacs Lisp code (see section "Byte Compilation" in *the Emacs Lisp Reference Manual*) and programs in C and other languages (see Section 23.1 [Compilation], page 199).

Complete Key

A complete key is a key sequence which fully specifies one action to be performed by Emacs. For example, `X` and `C-f` and `C-x m` are complete keys. Complete keys derive their meanings from being bound (q.v.) to commands (q.v.). Thus, `X` is conventionally bound to a command to insert '`X`' in the buffer; `C-x m` is conventionally bound to a command to begin composing a mail message. See Section 2.2 [Keys], page 20.

Completion

Completion is what Emacs does when it automatically fills out an abbreviation for a name into the entire name. Completion is done for minibuffer (q.v.) arguments when the set of possible valid inputs is known; for example, on command names, buffer names, and file names. Completion occurs when `TAB`, `SPC` or `RET` is typed. See Section 6.3 [Completion], page 37.

Continuation Line

When a line of text is longer than the width of the window, it takes up more than one screen line when displayed. We say that the text line is continued, and all screen lines used for it after the first are called continuation lines. See Chapter 4 [Basic Editing], page 25.

Control Character

ASCII characters with octal codes 0 through 037, and also code 0177, do not have graphic images assigned to them. These are the Control characters. To type a Control character, hold down the CTRL key and type the corresponding non-Control character. RET, TAB, ESC, LFD and DEL are all control characters. See Section 2.1 [User Input], page 19.

When you are using the X Window System, every non-control character has a corresponding control character variant.

Copyleft
: A copyleft is a notice giving the public legal permission to redistribute a program or other work of art. Copylefts are used by left-wing programmers to give people equal rights, just as copyrights are used by right-wing programmers to gain power over other people.

Current Buffer
: The current buffer in Emacs is the Emacs buffer on which most editing commands operate. You can select any Emacs buffer as the current one. See Chapter 16 [Buffers], page 119.

Current Line
: The line point is on (see Section 1.1 [Point], page 15).

Current Paragraph
: The paragraph that point is in. If point is between paragraphs, the current paragraph is the one that follows point. See Section 21.3 [Paragraphs], page 145.

Current Defun
: The defun (q.v.) that point is in. If point is between defuns, the current defun is the one that follows point. See Section 22.4 [Defuns], page 166.

Cursor
: The cursor is the rectangle on the screen which indicates the position called point (q.v.) at which insertion and deletion takes place. The cursor is on or under the character that follows point. Often people speak of 'the cursor' when, strictly speaking, they mean 'point'. See Chapter 4 [Basic Editing], page 25.

Customization
: Customization is making minor changes in the way Emacs works. It is often done by setting variables (see Section 31.2 [Variables], page 292) or by rebinding key sequences (see Section 31.4.1 [Keymaps], page 302).

Default Argument
: The default for an argument is the value that will be assumed if you do not specify one. When the minibuffer is used to read an argument, the default argument is used if you just type RET. See Chapter 6 [Minibuffer], page 35.

Default Directory
: When you specify a file name that does not start with '/' or '~', it is interpreted relative to the current buffer's default directory. See Section 6.1 [Minibuffer File], page 35.

Defun
: A defun is a list at the top level of parenthesis or bracket structure in a program. It is so named because most such lists in Lisp programs are calls to the Lisp function `defun`. See Section 22.4 [Defuns], page 166.

`DEL`
: `DEL` is a character that runs the command to delete one character of text. See Chapter 4 [Basic Editing], page 25.

Deletion
: Deletion means erasing text without copying it into the kill ring (q.v.). The alternative is killing (q.v.). See Section 10.1 [Killing], page 57.

Deletion of Files
: Deleting a file means erasing it from the file system. See Section 15.10 [Misc File Ops], page 116.

Deletion of Messages
: Deleting a message means flagging it to be eliminated from your mail file. Until you expunge (q.v.) the mail file, you can still undelete the messages you have deleted. See Section 27.3 [Rmail Deletion], page 229.

Deletion of Windows
: Deleting a window means eliminating it from the screen. Other windows expand to use up the space. The deleted window can never come back, but no actual text is thereby lost. See Chapter 17 [Windows], page 125.

Directory
: File directories are named collections in the file system, within which you can place individual files or subdirectories. See Section 15.8 [Directories], page 115.

Dired
: Dired is the Emacs facility that displays the contents of a file directory and allows you to "edit the directory", performing operations on the files in the directory. See Chapter 28 [Dired], page 241.

Disabled Command
: A disabled command is one that you may not run without special confirmation. The usual reason for disabling a command is that it is confusing for beginning users. See Section 31.4.9 [Disabling], page 310.

Down Event
: Short for 'button down event'.

Drag Event
: A drag event is the kind of input event generated when you press a mouse button, move the mouse, and then release the button. See Section 31.4.8 [Mouse Buttons], page 308.

Dribble File
: A file into which Emacs writes all the characters that the user types on the keyboard. Dribble files are used to make a record for debugging Emacs bugs. Emacs does not make a dribble file unless you tell it to. See Section 32.3 [Bugs], page 321.

Echo Area
: The echo area is the bottom line of the screen, used for echoing the arguments to commands, for asking questions, and printing brief messages (including error messages). See Section 1.2 [Echo Area], page 16.

Echoing
: Echoing is acknowledging the receipt of commands by displaying them (in the echo area). Emacs never echoes single-character key sequences; longer key sequences echo only if you pause while typing them.

Error
: An error occurs when an Emacs command cannot execute in the current circumstances. When an error occurs, execution of the command stops (unless the command has been programmed to do otherwise) and Emacs reports the error by printing an error message (q.v.). Type-ahead is discarded. Then Emacs is ready to read another editing command.

Error Messages
: Error messages are single lines of output printed by Emacs when the user asks for something impossible to do (such as, killing text forward when point is at the end of the buffer). They appear in the echo area, accompanied by a beep.

`ESC`
: `ESC` is a character used as a prefix for typing Meta characters on keyboards lacking a `META` key. Unlike the `META` key (which, like the `SHIFT` key, is held down while another character is typed), the `ESC` key is pressed once and applies to the next character typed.

Expunging
: Expunging a mail file or Dired buffer means really discarding the messages or files you have previously flagged for deletion.

Fill Prefix
: The fill prefix is a string that should be expected at the beginning of each line when filling is done. It is not regarded as part of the text to be filled. See Section 21.5 [Filling], page 147.

Filling
: Filling text means moving text from line to line so that all the lines are approximately the same length. See Section 21.5 [Filling], page 147.

Frame
: A frame is a rectangular cluster of Emacs windows. When using X Windows, you can create more than one Emacs frame, each having its own X window, and then you can subdivide each frame into Emacs windows as you wish. See Chapter 18 [Frames], page 129.

Function Key
: A function key is a key on the keyboard that does not correspond to any character. See Section 31.4.7 [Function Keys], page 306.

Global
: Global means 'independent of the current environment; in effect throughout Emacs'. It is the opposite of local (q.v.). Particular examples of the use of 'global' appear below.

Global Abbrev
: A global definition of an abbrev (q.v.) is effective in all major modes that do not have local (q.v.) definitions for the same abbrev. See Chapter 24 [Abbrevs], page 211.

Global Keymap
: The global keymap (q.v.) contains key bindings that are in effect except when overridden by local key bindings in a major mode's local keymap (q.v.). See Section 31.4.1 [Keymaps], page 302.

Global Substitution
: Global substitution means replacing each occurrence of one string by another string through a large amount of text. See Section 13.7 [Replace], page 84.

Global Variable
: The global value of a variable (q.v.) takes effect in all buffers that do not have their own local (q.v.) values for the variable. See Section 31.2 [Variables], page 292.

Graphic Character
: Graphic characters are those assigned pictorial images rather than just names. All the non-Meta (q.v.) characters except for the Control (q.v.) characters are graphic characters. These include letters, digits, punctuation, and spaces; they do not include `RET` or `ESC`. In Emacs, typing a graphic character inserts that character (in ordinary editing modes). See Chapter 4 [Basic Editing], page 25.

Hardcopy
: Hardcopy means printed output. Emacs has commands for making printed listings of text in Emacs buffers. See Section 30.5 [Hardcopy], page 283.

HELP
: You can type HELP at any time to ask what options you have, or to ask what any command does. The character HELP is really C-h. See Chapter 8 [Help], page 45.

Hyper
: Hyper is the name of a modifier bit which a keyboard input character may have. To make a character Hyper, type it while holding down the HYPER key. Such characters are given names that start with Hyper- (usually written H- for short). See Section 2.1 [User Input], page 19.

Inbox
: An inbox is a file in which mail is delivered by the operating system. Rmail transfers mail from inboxes to mail files (q.v.) in which the mail is then stored permanently or until explicitly deleted. See Section 27.4 [Rmail Inbox], page 230.

Indentation
: Indentation means blank space at the beginning of a line. Most programming languages have conventions for using indentation to illuminate the structure of the program, and Emacs has special commands to adjust indentation. See Chapter 20 [Indentation], page 139.

Insertion
: Insertion means copying text into the buffer, either from the keyboard or from some other place in Emacs.

Justification
: Justification means adding extra spaces to lines of text to make them come exactly to a specified width. See Section 21.5 [Filling], page 147.

Keyboard Macros
: Keyboard macros are a way of defining new Emacs commands from sequences of existing ones, with no need to write a Lisp program. See Section 31.3 [Keyboard Macros], page 298.

Key Sequence
: A key sequence (key, for short) is a sequence of characters that, when input to Emacs, is meaningful as a single unit. If the key sequence is enough to specify one action, it is a complete key (q.v.); if it is not enough, it is a prefix key (q.v.). See Section 2.2 [Keys], page 20.

Keymap
: The keymap is the data structure that records the bindings (q.v.) of key sequences to the commands that they run. For example, the global keymap binds the character C-n to the command function `next-line`. See Section 31.4.1 [Keymaps], page 302.

Keyboard Translation Table
: The keyboard translation table is an array that translates the character codes that come from the terminal into the character

codes that make up key sequences. See Section 31.5 [Keyboard Translations], page 310.

Kill Ring
: The kill ring is where all text you have killed recently is saved. You can reinsert any of the killed text still in the ring; this is called yanking (q.v.). See Section 10.2 [Yanking], page 59.

Killing
: Killing means erasing text and saving it on the kill ring so it can be yanked (q.v.) later. Some other systems call this "cutting". Most Emacs commands to erase text do killing, as opposed to deletion (q.v.). See Section 10.1 [Killing], page 57.

Killing Jobs
: Killing a job (such as, an invocation of Emacs) means making it cease to exist. Any data within it, if not saved in a file, is lost. See Section 3.1 [Exiting], page 23.

List
: A list is, approximately, a text string beginning with an open parenthesis and ending with the matching close parenthesis. In C mode and other non-Lisp modes, groupings surrounded by other kinds of matched delimiters appropriate to the language, such as braces, are also considered lists. Emacs has special commands for many operations on lists. See Section 22.2 [Lists], page 164.

Local
: Local means 'in effect only in a particular context'; the relevant kind of context is a particular function execution, a particular buffer, or a particular major mode. It is the opposite of 'global' (q.v.). Specific uses of 'local' in Emacs terminology appear below.

Local Abbrev
: A local abbrev definition is effective only if a particular major mode is selected. In that major mode, it overrides any global definition for the same abbrev. See Chapter 24 [Abbrevs], page 211.

Local Keymap
: A local keymap is used in a particular major mode; the key bindings (q.v.) in the current local keymap override global bindings of the same key sequences. See Section 31.4.1 [Keymaps], page 302.

Local Variable
: A local value of a variable (q.v.) applies to only one buffer. See Section 31.2.4 [Locals], page 295.

M-
: `M-` in the name of a character is an abbreviation for `META`, one of the modifier keys that can accompany any character. See Section 2.1 [User Input], page 19.

M-C- 'M-C-' in the name of a character is an abbreviation for Control-Meta; it means the same thing as 'C-M-'. If your terminal lacks a real META key, you type a Control-Meta character by typing ESC and then typing the corresponding Control character. See Section 2.1 [User Input], page 19.

M-x M-x is the key sequence which is used to call an Emacs command by name. This is how you run commands that are not bound to key sequences. See Chapter 7 [M-x], page 43.

Mail Mail means messages sent from one user to another through the computer system, to be read at the recipient's convenience. Emacs has commands for composing and sending mail, and for reading and editing the mail you have received. See Chapter 26 [Sending Mail], page 221. See Chapter 27 [Rmail], page 227, for how to read mail.

Mail File A mail file is a file which is edited using Rmail and in which Rmail stores mail. See Chapter 27 [Rmail], page 227.

Major Mode
The Emacs major modes are a mutually exclusive set of options, each of which configures Emacs for editing a certain sort of text. Ideally, each programming language has its own major mode. See Chapter 19 [Major Modes], page 137.

Mark The mark points to a position in the text. It specifies one end of the region (q.v.), point being the other end. Many commands operate on all the text from point to the mark. Each buffer has its own mark. See Chapter 9 [Mark], page 51.

Mark Ring
The mark ring is used to hold several recent previous locations of the mark, just in case you want to move back to them. Each buffer has its own mark ring. See Chapter 10 [Mark Ring], page 57.

Message See 'mail'.

Meta Meta is the name of a modifier bit which a command character may have. It is present in a character if the character is typed with the META key held down. Such characters are given names that start with Meta- (usually written M- for short). For example, M-< is typed by holding down META and at the same time typing < (which itself is done, on most terminals, by holding down SHIFT and typing ,). See Section 2.1 [User Input], page 19.

Meta Character
: A Meta character is one whose character code includes the Meta bit.

Minibuffer
: The minibuffer is the window that appears when necessary inside the echo area (q.v.), used for reading arguments to commands. See Chapter 6 [Minibuffer], page 35.

Minibuffer History
: The minibuffer history records the text you have specified in the past for minibuffer arguments, so you can conveniently use the same text again. See Section 6.4 [Minibuffer History], page 39.

Minor Mode
: A minor mode is an optional feature of Emacs which can be switched on or off independently of all other features. Each minor mode has a command to turn it on or off. See Section 31.1 [Minor Modes], page 291.

Minor Mode Keymap
: A keymap that belongs to a minor mode and is active when that mode is enabled. Minor mode keymaps take precedence over the buffer's local keymap, just as the local keymap takes precedence over the global keymap. See Section 31.4.1 [Keymaps], page 302.

Mode Line
: The mode line is the line at the bottom of each window (q.v.), giving status information on the buffer displayed in that window. See Chapter 2 [Mode Line], page 19.

Modified Buffer
: A buffer (q.v.) is modified if its text has been changed since the last time the buffer was saved (or since when it was created, if it has never been saved). See Section 15.3 [Saving], page 96.

Moving Text
: Moving text means erasing it from one place and inserting it in another. The usual way to move text by killing (q.v.) and then yanking (q.v.). See Section 10.1 [Killing], page 57.

Named Mark
: A named mark is a register (q.v.) in its role of recording a location in text so that you can move point to that location. See Chapter 11 [Registers], page 67.

Narrowing
: Narrowing means creating a restriction (q.v.) that limits editing in the current buffer to only a part of the text in the buffer. Text outside that part is inaccessible to the user until the boundaries are widened again, but it is still there, and saving the file saves it all. See Section 30.4 [Narrowing], page 282.

Newline
: LFD characters in the buffer terminate lines of text and are called newlines. See Section 2.4 [Text Characters], page 22.

Numeric Argument
: A numeric argument is a number, specified before a command, to change the effect of the command. Often the numeric argument serves as a repeat count. See Section 4.9 [Arguments], page 31.

Option
: An option is a variable (q.v.) that exists so that you can customize Emacs by giving it a new value. See Section 31.2 [Variables], page 292.

Overwrite Mode
: Overwrite mode is a minor mode. When it is enabled, ordinary text characters replace the existing text after point rather than pushing it to the right. See Section 31.1 [Minor Modes], page 291.

Page
: A page is a unit of text, delimited by formfeed characters (ASCII control-L, code 014) coming at the beginning of a line. Some Emacs commands are provided for moving over and operating on pages. See Section 21.4 [Pages], page 146.

Paragraphs
: Paragraphs are the medium-size unit of English text. There are special Emacs commands for moving over and operating on paragraphs. See Section 21.3 [Paragraphs], page 145.

Parsing
: We say that certain Emacs commands parse words or expressions in the text being edited. Really, all they know how to do is find the other end of a word or expression. See Section 31.6 [Syntax], page 311.

Point
: Point is the place in the buffer at which insertion and deletion occur. Point is considered to be between two characters, not at one character. The terminal's cursor (q.v.) indicates the location of point. See Chapter 4 [Basic], page 25.

Prefix Argument
: See 'numeric argument'.

Prefix Key
: A prefix key is a key sequence (q.v.) whose sole function is to introduce a set of longer key sequences. `C-x` is an example of prefix key; any two-character sequence starting with `C-x` is therefore a legitimate key sequence. See Section 2.2 [Keys], page 20.

Primary Mail File
: Your primary mail file is the file named '`RMAIL`' in your home directory, where all mail that you receive is stored by Rmail

unless you make arrangements to do otherwise. See Chapter 27 [Rmail], page 227.

Prompt
A prompt is text printed to ask the user for input. Printing a prompt is called prompting. Emacs prompts always appear in the echo area (q.v.). One kind of prompting happens when the minibuffer is used to read an argument (see Chapter 6 [Minibuffer], page 35); the echoing which happens when you pause in the middle of typing a multicharacter key sequence is also a kind of prompting (see Section 1.2 [Echo Area], page 16).

Quitting
Quitting means cancelling a partially typed command or a running command, using `C-g`. See Section 32.1 [Quitting], page 317.

Quoting
Quoting means depriving a character of its usual special significance. In Emacs this is usually done with `C-q`. What constitutes special significance depends on the context and on convention. For example, an "ordinary" character as an Emacs command inserts itself; so in this context, a special character is any character that does not normally insert itself (such as `DEL`, for example), and quoting it makes it insert itself as if it were not special. Not all contexts allow quoting. See Chapter 4 [Basic Editing], page 25.

Read-Only Buffer
A read-only buffer is one whose text you are not allowed to change. Normally Emacs makes buffers read-only when they contain text which has a special significance to Emacs; for example, Dired buffers. Visiting a file that is write protected also makes a read-only buffer. See Chapter 16 [Buffers], page 119.

Recursive Editing Level
A recursive editing level is a state in which part of the execution of a command involves asking the user to edit some text. This text may or may not be the same as the text to which the command was applied. The mode line indicates recursive editing levels with square brackets ('`[`' and '`]`'). See Section 30.9 [Recursive Edit], page 287.

Redisplay
Redisplay is the process of correcting the image on the screen to correspond to changes that have been made in the text being edited. See Chapter 1 [Screen], page 15.

Regexp
See 'regular expression'.

Region
The region is the text between point (q.v.) and the mark (q.v.). Many commands operate on the text of the region. See Chapter 9 [Mark], page 51.

Registers
: Registers are named slots in which text or buffer positions or rectangles can be saved for later use. See Chapter 11 [Registers], page 67.

Regular Expression
: A regular expression is a pattern that can match various text strings; for example, '`l[0-9]+`' matches '`l`' followed by one or more digits. See Section 13.5 [Regexps], page 79.

Repeat Count
: See 'numeric argument'.

Replacement
: See 'global substitution'.

Restriction
: A buffer's restriction is the amount of text, at the beginning or the end of the buffer, that is temporarily inaccessible. Giving a buffer a nonzero amount of restriction is called narrowing (q.v.). See Section 30.4 [Narrowing], page 282.

`RET`
: `RET` is a character than in Emacs runs the command to insert a newline into the text. It is also used to terminate most arguments read in the minibuffer (q.v.). See Section 2.1 [User Input], page 19.

Saving
: Saving a buffer means copying its text into the file that was visited (q.v.) in that buffer. This is the way text in files actually gets changed by your Emacs editing. See Section 15.3 [Saving], page 96.

Scrolling
: Scrolling means shifting the text in the Emacs window so as to see a different part of the buffer. See Chapter 12 [Display], page 69.

Searching
: Searching means moving point to the next occurrence of a specified string. See Chapter 13 [Search], page 75.

Selecting
: Selecting a buffer means making it the current (q.v.) buffer. See Chapter 16 [Buffers], page 119.

Self-Documentation
: Self-documentation is the feature of Emacs which can tell you what any command does, or give you a list of all commands related to a topic you specify. You ask for self-documentation with the help character, `C-h`. See Chapter 8 [Help], page 45.

Sentences
: Emacs has commands for moving by or killing by sentences. See Section 21.2 [Sentences], page 144.

Sexp
: A sexp (short for 's-expression') is the basic syntactic unit of Lisp in its textual form: either a list, or Lisp atom. Many Emacs commands operate on sexps. The term 'sexp' is generalized to languages other than Lisp, to mean a syntactically recognizable expression. See Section 22.2 [Lists], page 164.

Simultaneous Editing
: Simultaneous editing means two users modifying the same file at once. Simultaneous editing if not detected can cause one user to lose his work. Emacs detects all cases of simultaneous editing and warns the user to investigate them. See Section 15.3.2 [Simultaneous Editing], page 101.

String
: A string is a kind of Lisp data object which contains a sequence of characters. Many Emacs variables are intended to have strings as values. The Lisp syntax for a string consists of the characters in the string with a '`"`' before and another '`"`' after. A '`"`' that is part of the string must be written as '`\"`' and a '`\`' that is part of the string must be written as '`\\`'. All other characters, including newline, can be included just by writing them inside the string; however, escape sequences as in C, such as '`\n`' for newline or '`\241`' using an octal character code, are allowed as well.

String Substitution
: See 'global substitution'.

Syntax Table
: The syntax table tells Emacs which characters are part of a word, which characters balance each other like parentheses, etc. See Section 31.6 [Syntax], page 311.

Super
: Super is the name of a modifier bit which a keyboard input character may have. To make a character Super, type it while holding down the `SUPER` key. Such characters are given names that start with `Super-` (usually written `s-` for short). See Section 2.1 [User Input], page 19.

Tag Table
: A tag table is a file that serves as an index to the function definitions in one or more other files. See Section 22.12 [Tags], page 179.

Termscript File
: A termscript file contains a record of all characters sent by Emacs to the terminal. It is used for tracking down bugs in Emacs redisplay. Emacs does not make a termscript file unless you tell it to. See Section 32.3 [Bugs], page 321.

Text
: Two meanings (see Chapter 21 [Text], page 143):

- Data consisting of a sequence of characters, as opposed to binary numbers, images, graphics commands, executable programs, and the like. The contents of an Emacs buffer are always text in this sense.
- Data consisting of written human language, as opposed to programs, or following the stylistic conventions of human language.

Top Level
: Top level is the normal state of Emacs, in which you are editing the text of the file you have visited. You are at top level whenever you are not in a recursive editing level (q.v.) or the minibuffer (q.v.), and not in the middle of a command. You can get back to top level by aborting (q.v.) and quitting (q.v.). See Section 32.1 [Quitting], page 317.

Transposition
: Transposing two units of text means putting each one into the place formerly occupied by the other. There are Emacs commands to transpose two adjacent characters, words, sexps (q.v.) or lines (see Section 14.2 [Transpose], page 89).

Truncation
: Truncating text lines in the display means leaving out any text on a line that does not fit within the right margin of the window displaying it. See also 'continuation line'. See Chapter 4 [Basic Editing], page 25.

Undoing
: Undoing means making your previous editing go in reverse, bringing back the text that existed earlier in the editing session. See Chapter 5 [Undo], page 33.

Variable
: A variable is an object in Lisp that can store an arbitrary value. Emacs uses some variables for internal purposes, and has others (known as 'options' (q.v.)) just so that you can set their values to control the behavior of Emacs. The variables used in Emacs that you are likely to be interested in are listed in the Variables Index in this manual. See Section 31.2 [Variables], page 292, for information on variables.

Version Control
: Version control systems keep track of multiple versions of a source file. They provide a more powerful alternative to keeping backup files (q.v.). See Section 15.7 [Version Control], page 106.

Visiting
: Visiting a file means loading its contents into a buffer (q.v.) where they can be edited. See Section 15.2 [Visiting], page 94.

Whitespace
: Whitespace is any run of consecutive formatting characters (space, tab, newline, and backspace).

Widening
: Widening is removing any restriction (q.v.) on the current buffer; it is the opposite of narrowing (q.v.). See Section 30.4 [Narrowing], page 282.

Window
: Emacs divides a frame (q.v.) into one or more windows, each of which can display the contents of one buffer (q.v.) at any time. See Chapter 1 [Screen], page 15, for basic information on how Emacs uses the screen. See Chapter 17 [Windows], page 125, for commands to control the use of windows.

Word Abbrev
: Synonymous with 'abbrev'.

Word Search
: Word search is searching for a sequence of words, considering the punctuation between them as insignificant. See Section 13.3 [Word Search], page 78.

Yanking
: Yanking means reinserting text previously killed. It can be used to undo a mistaken kill, or for copying or moving text. Some other systems call this "pasting". See Section 10.2 [Yanking], page 59.

Key (Character) Index

D

E

F

G

Command and Function Index

D

E

F

G

H

N

O

P

T

Variable Index

N

O

P

R

S

T

U

V

W

X

Concept Index

Short Contents

Table of Contents