XEmacs User's Manual

July 1994
(General Public License upgraded, January 1991)

Richard Stallman

Lucid, Inc.

and

Ben Wing

Preface

This manual documents the use and simple customization of the XEmacs editor. The reader is not expected to be a programmer to use this editor, and simple customizations do not require programming skills either. Users who are not interested in customizing XEmacs can ignore the scattered customization hints.

This document is primarily a reference manual, but it can also be used as a primer. However, if you are new to XEmacs, consider using the on-line, learn-by-doing tutorial, which you get by running XEmacs and typing *C-h t*. With it, you learn XEmacs by using XEmacs on a specially designed file which describes commands, tells you when to try them, and then explains the results you see. Using the tutorial gives a more vivid introduction than the printed manual. Also consider reading the XEmacs New User's Guide, which is intended specifically as an introductory manual rather than as a reference guide.

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

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

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

XEmacs 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 manual is for XEmacs installed on UNIX systems. XEmacs also exists on Microsoft Windows and Windows NT as Win-Emacs (which is actually based on Lucid Emacs 19.6, an older incarnation of XEmacs).

GNU GENERAL PUBLIC LICENSE

Version 1, February 1989

Preamble

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

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

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

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

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

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

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

TERMS AND CONDITIONS

1. This License Agreement applies to any program or other work which contains a notice placed by the copyright holder saying it may be distributed under the terms of this General Public License. The "Program", below, refers to any such program or work, and a "work based on the Program" means either the Program or any work containing the Program or a portion of it, either verbatim or with modifications. Each licensee is addressed as "you".

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

   Mere aggregation of another independent work with the Program (or its derivative) on a volume of a storage or distribution medium does not bring the other work under the scope of these terms.
4. You may copy and distribute the Program (or a portion or derivative of it, under Paragraph 2) in object code or executable form under the terms of Paragraphs 1 and 2 above provided that you also do one of the following:
   - accompany it with the complete corresponding machine-readable source code, which must be distributed under the terms of Paragraphs 1 and 2 above; or,
   - accompany it with a written offer, valid for at least three years, to give any third party free (except for a nominal charge for the cost of distribution) a complete machine-readable copy of the corresponding source code, to be distributed under the terms of Paragraphs 1 and 2 above; or,
   - accompany it with the information you received as to where the corresponding source code may be obtained. (This alternative is allowed only for noncommercial distribution and only if you received the program in object code or executable form alone.)

   Source code for a work means the preferred form of the work for making modifications to it. For an executable file, complete source code means all the source code for all modules it contains; but, as a special exception, it need not include source code for modules which are standard libraries that accompany the operating system on which the executable file runs, or for standard header files or definitions files that accompany that operating system.
5. You may not copy, modify, sublicense, distribute or transfer the Program except as expressly provided under this General Public License. Any attempt otherwise to copy, modify, sublicense, distribute or transfer the Program is void, and will automatically terminate your rights to use the Program under this License. However, parties who have received copies, or rights to use copies, from you under this General Public License will not have their licenses terminated so long as such parties remain in full compliance.

6. By copying, distributing or modifying the Program (or any work based on the Program) you indicate your acceptance of this license to do so, and all its terms and conditions.
7. Each time you redistribute the Program (or any work based on the Program), the recipient automatically receives a license from the original licensor to copy, distribute or modify the Program subject to these terms and conditions. You may not impose any further restrictions on the recipients' exercise of the rights granted herein.

8. The Free Software Foundation may publish revised and/or new versions of the General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

   Each version is given a distinguishing version number. If the Program specifies a version number of the license which applies to it and "any later version", you have the option of following the terms and conditions either of that version or of any later version published by the Free Software Foundation. If the Program does not specify a version number of the license, you may choose any version ever published by the Free Software Foundation.
9. If you wish to incorporate parts of the Program into other free programs whose distribution conditions are different, write to the author to ask for permission. For software which is copyrighted by the Free Software Foundation, write to the Free Software Foundation; we sometimes make exceptions for this. Our decision will be guided by the two goals of preserving the free status of all derivatives of our free software and of promoting the sharing and reuse of software generally.

NO WARRANTY

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

END OF TERMS AND CONDITIONS

Appendix: How to Apply These Terms to Your New Programs

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

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

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

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

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

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

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

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

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

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

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

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

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

That's all there is to it!

Distribution

XEmacs is *free*; this means that everyone is free to use it and free to redistribute it on a free basis. XEmacs 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 XEmacs that they might get from you. The precise conditions are found in the GNU General Public License that comes with XEmacs and also appears following this section.

The easiest way to get a copy of XEmacs 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 version of XEmacs from the anonymous FTP server '`ftp.xemacs.org`' in the directory '`/pub/xemacs`'. It can also be found at numerous other archive sites around the world; check the file '`etc/DISTRIB`' in an XEmacs distribution for the latest known list.

Getting Other Versions of Emacs

The Free Software Foundation's version of Emacs (called *FSF Emacs* in this manual and often referred to as *GNU Emacs*) is available by anonymous FTP from '`prep.ai.mit.edu`'.

Win-Emacs, an older version of XEmacs that runs on Microsoft Windows and Windows NT, is available by anonymous FTP from '`ftp.netcom.com`' in the directory '`/pub/pe/pearl`', or from '`ftp.cica.indiana.edu`' as the files '`wemdemo*.zip`' in the directory '`/pub/pc/win3/demo`'.

Introduction

You are reading about XEmacs, an incarnation of the advanced, self-documenting, customizable, extensible real-time display editor Emacs. XEmacs provides many powerful display and user-interface capabilities not found in other Emacsen and is mostly upwardly compatible with GNU Emacs from the Free Software Foundation (referred to as *FSF Emacs* in this manual). XEmacs also comes standard with a great number of useful packages.

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

We call XEmacs 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 41.

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

Self-documenting means that at any time you can type a special character, `Control-h`, to find out what your options are. You can also use `C-h` to find out what a command does, or to find all the commands relevant to a topic. See Chapter 8 [Help], page 59.

Customizable means you can change the definitions of XEmacs commands. For example, if you use a programming language in which comments start with '`<**`' and end with '`**>`', you can tell the XEmacs comment manipulation commands to use those strings (see Section 22.6 [Comments], page 172). Another sort of customization is rearrangement of the command set. For example, you can set up the four basic cursor motion commands (up, down, left and right) on keys in a diamond pattern on the keyboard if you prefer. See Chapter 29 [Customization], page 257.

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

1 The XEmacs Frame

Frame
: In many environments, such as a tty terminal, an XEmacs frame literally takes up the whole screen. If you are running XEmacs in a multi-window system like the X Window System, the XEmacs frame takes up one X window. See Section 1.4 [XEmacs under X], page 16, for more information.

Window
: No matter what environment you are running in, XEmacs allows you to look at several buffers at the same time by having several windows be part of the frame. Often, the whole frame is taken up by just one window, but you can split the frame into two or more subwindows. If you are running XEmacs under the X window system, that means you can have several *XEmacs windows* inside the X window that contains the XEmacs frame. You can even have multiple frames in different X windows, each with their own set of subwindows.

Each XEmacs frame displays a variety of information:

- The biggest area usually displays the text you are editing. It may consist of one window or of two or more windows if you need to look at two buffers a the same time.
- Below each text window's last line is a *mode line* (see Section 1.3 [Mode Line], page 15), which describes what is going on in that window. The mode line is in inverse video if the terminal supports that. If there are several XEmacs windows in one frame, each window has its own mode line.
- At the bottom of each XEmacs frame is the *echo area* or *minibuffer window*(see Section 1.2 [Echo Area], page 14). It is used by XEmacs to exchange information with the user. There is only one echo area per XEmacs frame.
- If you are running XEmacs under the X Window System, a menu bar at the top of the frame makes shortcuts to several of the commands available (see Section 2.4 [Pull-down Menus], page 25).

You can subdivide the XEmacs frame into multiple text windows, and use each window for a different file (see Chapter 17 [Windows], page 131). Multiple XEmacs windows are tiled vertically on the XEmacs frame. The upper XEmacs window is separated from the lower window by its mode line.

When there are multiple, tiled XEmacs windows on a single XEmacs frame, the XEmacs window receiving input from the keyboard has the *keyboard focus* and is called the *selected window*. The selected window contains the cursor, which indicates the insertion point. If you are working in an environment that permits multiple XEmacs frames, and you move the focus from one XEmacs frame into another, the selected window is the one that was last selected in that frame.

The same text can be displayed simultaneously in several XEmacs windows, which can be in different XEmacs frames. If you alter the text in an XEmacs buffer by editing it in one XEmacs window, the changes are visible in all XEmacs windows containing that buffer.

1.1 Point

When XEmacs is running, the cursor shows the location at which editing commands will take effect. This location is called *point*. You can use keystrokes or the mouse cursor to move point through the text and edit the text at different places.

While the cursor appears to point *at* a character, you should think of point as *between* two characters: it points *before* the character on which the cursor appears. The exception is at the end of the line, where the cursor appears after the last character of the line. Where the display is capable, the cursor at the end of the line will appear differently from a cursor over whitespace at the end of the line. (In an X Windows frame, the end-of-line cursor is half the width of a within-line cursor.) Sometimes people speak of "the cursor" when they mean "point," or speak of commands that move point as "cursor motion" commands.

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

If you are editing several files in XEmacs, each file has its own point location. A file that is not being displayed remembers where point is. Point becomes visible at the correct location when you look at the file again.

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

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

1.2 The Echo Area

The line at the bottom of the frame (below the mode line) is the *echo area*. XEmacs uses this area to communicate with the user:

- *Echoing* means printing out the characters that the user types. XEmacs never echoes single-character commands. Multi-character commands are echoed only if you pause while typing them: As soon as you pause for more than one second in the middle of a command, all the characters of the command so far are echoed. This is intended to *prompt* you for the rest of the command. Once echoing has started, the rest of the command is echoed immediately 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 12.4 [Display Vars], page 83).
- If you issue a command that cannot be executed, XEmacs may print an *error message* in the echo area. Error messages are accompanied by a beep or by flashing the frame. 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 similar to error messages, but are not announced with a beep and do not throw away input. Sometimes a message tells you what the command has done, when this is not obvious from looking at the text being edited. Sometimes the sole purpose of a command is to print a message giving you

specific information. For example, the command `C-x =` is used to print a message describing the character position of point in the text and its current column in the window. Commands that take a long time often display messages ending in '`...`' while they are working, and add '`done`' at the end when they are finished.

- The echo area is also used to display the *minibuffer*, a window that is used for reading arguments to commands, such as the name of a file to be edited. When the minibuffer is in use, the echo area displays with a prompt string that usually ends with a colon. The cursor appears after the prompt. You can always get out of the minibuffer by typing `C-g`. See Chapter 6 [Minibuffer], page 51.

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 '`XEmacs:` *something*'.

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

Normally, the mode line has the following appearance:

`--`*ch*`-XEmacs:` *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 two dashes ('`--`') if the buffer has not been edited. Exception: for a read-only buffer, it is '`%%`'.

buf is the name of the window's chosen *buffer*. The chosen buffer in the selected window (the window that the cursor is in) is also XEmacs's selected buffer, the buffer in which editing takes place. When we speak of what some command does to "the buffer", we mean the currently selected buffer. See Chapter 16 [Buffers], page 125.

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

major is the name of the *major mode* in effect in the buffer. At any time, each buffer is in one and only one major mode. The available major modes include Fundamental mode (the least specialized), Text mode, Lisp mode, and C mode. See Chapter 19 [Major Modes], page 143, for details on how the modes differ and how you select one.

minor is a list of some of the *minor modes* that are turned on in the window's chosen buffer. For example, '`Fill`' means that Auto Fill mode is on. `Abbrev` means that Word Abbrev mode is on. `Ovwrt` means that Overwrite mode is on. See Section 29.1 [Minor Modes], page 257, 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 28.3 [Narrowing], page 252. `Def` means that a keyboard macro is being defined. See Section 29.3 [Keyboard Macros], page 267.

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.

If XEmacs is currently inside a recursive editing level, square brackets ('`[...]`') appear around the parentheses that surround the modes. If XEmacs is in one recursive editing level within another, double square brackets appear, and so on. Since information on recursive editing applies to XEmacs in general and not to any one buffer, the square brackets appear in every mode line on the screen or not in any of them. See Section 28.5 [Recursive Edit], page 253.

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

    hh:mmpm l.ll [d]

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

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

Customization note: the variable `mode-line-inverse-video` controls whether the mode line is displayed in inverse video (assuming the terminal supports it); `nil` means no inverse video. The default is `t`. For X frames, simply set the foreground and background colors appropriately.

1.4 Using XEmacs Under the X Window System

XEmacs can be used with the X Window System and a window manager like MWM or TWM. In that case, the X window manager opens, closes, and resizes XEmacs frames. You use the window manager's mouse gestures to perform the operations. Consult your window manager guide or reference manual for information on manipulating X windows.

When you are working under X, each X window (that is, each XEmacs frame) has a menu bar for mouse-controlled operations (see Section 2.4 [Pull-down Menus], page 25).

XEmacs under X is also a multi-frame XEmacs. You can use the **New Frame** menu item from the **File** menu to create a new XEmacs frame in a new X window from the same process. The different frames will share the same buffer list, but you can look at different buffers in the different frames.

The function `find-file-other-frame` is just like `find-file`, but creates a new frame to display the buffer in first. This is normally bound to *C-x 5 C-f*, and is what the **Open File, New Frame** menu item does.

The function `switch-to-buffer-other-frame` is just like `switch-to-buffer`, but creates a new frame to display the buffer in first. This is normally bound to *C-x 5 b*.

You can specify a different default frame size other than the one provided. Use the variable `default-frame-alist`, which is an alist of default values for frame creation other than the first one. These may be set in your init file, like this:

```
(setq default-frame-alist '((width . 80) (height . 55)))
```

For values specific to the first XEmacs frame, you must use X resources. The variable `x-frame-defaults` takes an alist of default frame creation parameters for X window frames. These override what is specified in '`~/.Xdefaults`' but are overridden by the arguments to the particular call to `x-create-frame`.

When you create a new frame, the variable `create-frame-hook` is called with one argument, the frame just created.

If you want to close one or more of the X windows you created using **New Frame**, use the **Delete Frame** menu item from the **File** menu.

If you are working with multiple frames, some special information applies:

- Two variables, `frame-title-format` and `frame-icon-title-format` determine the title of the frame and the title of the icon that results if you shrink the frame.
- The variables `auto-lower-frame` and `auto-raise-frame` position a frame. If true, `auto-lower-frame` lowers a frame to the bottom when it is no longer selected. If true, `auto-raise-frame` raises a frame to the top when it is selected. Under X, most ICCCM-compliant window managers will have options to do this for you, but these variables are provided in case you are using a broken window manager.
- There is a new frame/modeline format directive, %S, which expands to the name of the current frame (a frame's name is distinct from its title; the name is used for resource lookup, among other things, and the title is simply what appears above the window.)

2 Keystrokes, Key Sequences, and Key Bindings

This chapter discusses the character set Emacs uses for input commands and inside files. You have already learned that the more frequently used Emacs commands are bound to keys. For example, `Control-f` is bound to `forward-char`. The following issues are covered:

- How keystrokes can be represented
- How you can create key sequences from keystrokes
- How you can add to the available modifier keys by customizing your keyboard: for example, you could have the ⟨Capslock⟩ key be understood as the ⟨Super⟩ key by Emacs. A ⟨Super⟩ key is used like ⟨Control⟩ or ⟨Meta⟩ in that you hold it while typing another key.

You will also learn how to customize existing key bindings and create new ones.

2.1 Keystrokes as Building Blocks of Key Sequences

Earlier versions of Emacs used only the ASCII character set, which defines 128 different character codes. Some of these codes are assigned graphic symbols like '`a`' and '`=`'; the rest are control characters, such as `Control-a` (also called `C-a`). `C-a` means you hold down the ⟨CTRL⟩ key and then press `a`.

Keybindings in XEmacs are not restricted to the set of keystrokes that can be represented in ASCII. XEmacs can tell the difference between, for example, `Control-h`, `Control-Shift-h`, and `Backspace`.

A keystroke is like a piano chord: you get it by simultaneously striking several keys. To be more precise, a keystroke consists of a possibly empty set of modifiers followed by a single *keysym*. The set of modifiers is small; it consists of `Control`, `Meta`, `Super`, `Hyper`, and `Shift`.

The rest of the keys on your keyboard, along with the mouse buttons, make up the set of keysyms. A keysym is usually what is printed on the keys on your keyboard. Here is a table of some of the symbolic names for keysyms:

`a,b,c...` alphabetic keys

`f1,f2...` function keys

`button1` left mouse button

`button2` middle mouse button

`button3` right mouse button

`button1up`
    upstroke on the left mouse button

`button2up`
    upstroke on the middle mouse button

`button3up`
    upstroke on the right mouse button

`return` Return key

Use the variable `keyboard-translate-table` only if you are on a dumb tty, as it cannot handle input that cannot be represented as ASCII. The value of this variable is a string used as a translate table for keyboard input or `nil`. Each character is looked up in this string and the contents used instead. If the string is of length `n`, character codes `N` and up are untranslated. If you are running Emacs under X, you should do the translations with the `xmodmap` program instead.

2.1.1 Representing Keystrokes

XEmacs represents keystrokes as lists. Each list consists of an arbitrary combination of modifiers followed by a single keysym at the end of the list. If the keysym corresponds to an ASCII character, you can use its character code. (A keystroke may also be represented by an event object, as returned by the `read-key-sequence` function; non-programmers need not worry about this.)

The following table gives some examples of how to list representations for keystrokes. Each list consists of sets of modifiers followed by keysyms:

(control a)
: Pressing ⟨CTRL⟩ and `a` simultaneously.

(control ?a)
: Another way of writing the keystroke *C-a*.

(control 65)
: Yet another way of writing the keystroke *C-a*.

(break)
: Pressing the ⟨BREAK⟩ key.

(control meta button2up)
: Release the middle mouse button, while pressing ⟨CTRL⟩ and ⟨META⟩.

Note: As you define keystrokes, you can use the *shift* key only as a modifier with characters that do not have a second keysym on the same key, such as *backspace* and *tab*. It is an error to define a keystroke using the ⟨shift⟩ modifier with keysyms such as `a` and `=`. The correct forms are *A* and `+`.

2.1.2 Representing Key Sequences

A *complete key sequence* is a sequence of keystrokes that Emacs understands as a unit. Key sequences are significant because you can bind them to commands. Note that not all sequences of keystrokes are possible key sequences. In particular, the initial keystrokes in a key sequence must make up a *prefix key sequence*.

Emacs represents a key sequence as a vector of keystrokes. Thus, the schematic representation of a complete key sequence is as follows:

```
[(modifier .. modifer keysym) ... (modifier .. modifier keysym)]
```

Here are some examples of complete key sequences:

[(control c) (control a)]
: Typing *C-c* followed by *C-a*

[(control c) (control 65)]
Typing *C-c* followed by *C-a*. (Using the ASCII code for the character 'a')

[(control c) (break)]
Typing *C-c* followed by the *break* character.

A *prefix key sequence* is the beginning of a series of longer sequences that are valid key sequences; adding any single keystroke to the end of a prefix results in a valid key sequence. For example, *control-x* is standardly defined as a prefix. Thus there is a two-character key sequence starting with *C-x* for each valid keystroke, giving numerous possibilities. Here are some samples:

- *[(control x) (c)]*
- *[(control x) (control c)]*

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

By contrast, the two-character sequence *[(control f) (control k)]* is not a key, because the *(control f)* is a complete key sequence in itself. You cannot give *[(control f (control k)]* an independent meaning as a command while *(control f)* is a complete sequence, because Emacs would understand ⟨C-f C-k⟩ as two commands.

The predefined prefix key sequences in Emacs are *(control c)*, *(control x)*, *(control h)*, *[(control x) (\4)]*, and *escape*. You can customize Emacs and could make new prefix keys or eliminate the default key sequences. See Section 29.4 [Key Bindings], page 269. For example, if you redefine *(control f)* as a prefix, *[(control f) (control k)]* automatically becomes a valid key sequence (complete, unless you define it as a prefix as well). Conversely, if you remove the prefix definition of *[(control x) (\4)]*, *[(control x) (\4) (f)]* (or *[(control x) (\4) anything]*) is no longer a valid key sequence.

Note that the above paragraphs uses \4 instead of simply 4, because \4 is the symbol whose name is "4", and plain 4 is the integer 4, which would have been interpreted as the ASCII value. Another way of representing the symbol whose name is "4" is to write ?4, which would be interpreted as the number 52, which is the ASCII code for the character "4". We could therefore actually have written 52 directly, but that is far less clear.

2.1.3 String Key Sequences

For backward compatibility, you may also represent a key sequence using strings. For example, we have the following equivalent representations:

`"\C-c\C-c"`
`[(control c) (control c)]`

`"\e\C-c"` `[(meta control c)]`

2.1.4 Assignment of the ⟨META⟩ Key

Not all terminals have the complete set of modifiers. Terminals that have a ⟨Meta⟩ key allow you to type Meta characters by just holding that key down. To type ***Meta-a***, hold down ⟨META⟩ and press `a`. On those terminals, the ⟨META⟩ key works 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⟩. 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.

If you are running under X and do not have a ⟨META⟩ key, it is possible to reconfigure some other key to be a ⟨META⟩ key. See Section 2.1.5 [Super and Hyper Keys], page 22.

Emacs believes the terminal has a ⟨META⟩ key if the variable `meta-flag` is non-`nil`. Normally this is set automatically according to the termcap entry for your terminal type. However, sometimes the termcap entry is wrong, and then it is useful to set this variable yourself. See Section 29.2 [Variables], page 257, for how to do this.

Note: If you are running under the X window system, the setting of the `meta-flag` variable is irrelevant.

2.1.5 Assignment of the ⟨SUPER⟩ and ⟨HYPER⟩ Keys

Most keyboards do not, by default, have ⟨SUPER⟩ or ⟨HYPER⟩ modifier keys. Under X, you can simulate the ⟨SUPER⟩ or ⟨HYPER⟩ key if you want to bind keys to sequences using ***super*** and ***hyper***. You can use the `xmodmap` program to do this.

For example, to turn your ⟨CAPS-LOCK⟩ key into a ⟨SUPER⟩ key, do the following:

Create a file called `~/.xmodmap`. In this file, place the lines

```
remove Lock = Caps_Lock
keysym Caps_Lock = Super_L
add Mod2 = Super_L
```

The first line says that the key that is currently called `Caps_Lock` should no longer behave as a "lock" key. The second line says that this should now be called `Super_L` instead. The third line says that the key called `Super_L` should be a modifier key, which produces the `Mod2` modifier.

To create a ⟨META⟩ or ⟨HYPER⟩ key instead of a ⟨SUPER⟩ key, replace the word `Super` above with `Meta` or `Hyper`.

Just after you start up X, execute the command `xmodmap /.xmodmap`. You can add this command to the appropriate initialization file to have the command executed automatically.

If you have problems, see the documentation for the xmodmap program. The X keyboard model is quite complicated, and explaining it is beyond the scope of this manual. However, we reprint the following description from the X Protocol document for your convenience:

A list of keysyms is associated with each keycode. If that list (ignoring trailing NoSymbol entries) is a single keysym 'K', then the list is treated as if it were the list ''K NoSymbol K NoSymbol''. If the list (ignoring trailing NoSymbol entries) is a pair of keysyms 'K1 K2', then the list is treated as if it were the list ''K1 K2 K1 K2''. If the list (ignoring trailing NoSymbol entries) is a triple of keysyms 'K1 K2 K3', then the list is treated as if it were the list ''K1 K2 K3 NoSymbol''.

The first four elements of the list are split into two groups of keysyms. Group 1 contains the first and second keysyms; Group 2 contains third and fourth keysyms. Within each group, if the second element of the group is NoSymbol, then the group should be treated as if the second element were the same as the first element, except when the first element is an alphabetic keysym 'K' for which both lowercase and uppercase forms are defined. In that case, the group should be treated as if the first element were the lowercase form of 'K' and the second element were the uppercase form of 'K'.

The standard rules for obtaining a keysym from a KeyPress event make use of only the Group 1 and Group 2 keysyms; no interpretation of other keysyms in the list is given here. (That is, the last four keysyms are unused.)

Which group to use is determined by modifier state. Switching between groups is controlled by the keysym named Mode_switch. Attach that keysym to some keycode and attach that keycode to any one of the modifiers Mod1 through Mod5. This modifier is called the *group modifier*. For any keycode, Group 1 is used when the group modifier is off, and Group 2 is used when the group modifier is on.

Within a group, which keysym to use is also determined by modifier state. The first keysym is used when the Shift and Lock modifiers are off. The second keysym is used when the Shift modifier is on, or when the Lock modifier is on and the second keysym is uppercase alphabetic, or when the Lock modifier is on and is interpreted as ShiftLock. Otherwise, when the Lock modifier is on and is interpreted as CapsLock, the state of the Shift modifier is applied first to select a keysym, but if that keysym is lower-case alphabetic, then the corresponding upper-case keysym is used instead.

In addition to the above information on keysyms, we also provide the following description of modifier mapping from the InterClient Communications Conventions Manual:

X11 supports 8 modifier bits, of which 3 are pre-assigned to Shift, Lock, and Control. Each modifier bit is controlled by the state of a set of keys, and these sets are specified in a table accessed by GetModifierMapping() and SetModifierMapping().

A client needing to use one of the pre-assigned modifiers should assume that the modifier table has been set up correctly to control these modifiers. The Lock modifier should be interpreted as Caps Lock or Shift Lock according to whether the keycodes in its controlling set include XK_Caps_Lock or XK_Shift_Lock.

Clients should determine the meaning of a modifier bit from the keysyms being used to control it.

A client needing to use an extra modifier, for example Meta, should:

1. Scan the existing modifier mappings.
   1. If it finds a modifier that contains a keycode whose set of keysyms includes `XK_Meta_L` or `XK_Meta_R`, it should use that modifier bit.
   2. If there is no existing modifier controlled by `XK_Meta_L` or `XK_Meta_R`, it should select an unused modifier bit (one with an empty controlling set) and:
2. If there is a keycode with `XL_Meta_L` in its set of keysyms, add that keycode to the set for the chosen modifier, and then:
   1. If there is a keycode with `XL_Meta_R` in its set of keysyms, add that keycode to the set for the chosen modifier, and then:
   2. If the controlling set is still empty, interact with the user to select one or more keys to be `Meta`.
3. If there are no unused modifier bits, ask the user to take corrective action.

This means that the `Mod1` modifier does not necessarily mean `Meta`, although some applications (such as twm and emacs 18) assume that. Any of the five unassigned modifier bits could mean `Meta`; what matters is that a modifier bit is generated by a keycode which is bound to the keysym `Meta_L` or `Meta_R`.

Therefore, if you want to make a ⟨META⟩ key, the right way is to make the keycode in question generate both a `Meta` keysym and some previously-unassigned modifier bit.

2.2 Representation of Characters

This section briefly discusses how characters are represented in Emacs buffers. See Section 2.1.2 [Key Sequences], page 20, for information on representing key sequences to create key bindings.

ASCII graphic characters in Emacs buffers are displayed with their graphics. ⟨LFD⟩ is the same as a newline character; it is displayed by starting a new line. ⟨TAB⟩ is displayed by moving to the next tab stop column (usually every 8 spaces). Other control characters are displayed as a caret ('`^`') followed by the non-control version of the character; thus, *`C-a`* is displayed as '`^A`'. Non-ASCII characters 128 and up are displayed with octal escape sequences; thus, character code 243 (octal), also called *`M-#`* when used as an input character, is displayed as '`\243`'.

The variable `ctl-arrow` may be used to alter this behavior. See Section 12.4 [Display Vars], page 83.

2.3 Keys and Commands

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

A function is a Lisp object that can be executed as a program. Usually it is a Lisp symbol that has been given a function definition; every symbol has a name, usually made of a few English words separated by dashes, such as `next-line` or `forward-word`. It also has a *definition*, which is a Lisp program. Only some functions can be the bindings of keys; these are functions whose definitions use `interactive` to specify how to call them interactively. Such functions are called *commands*, and their names are *command names*. More information on this subject will appear in the *XEmacs Lisp Reference Manual*.

The bindings between keys and functions are recorded in various tables called *keymaps*. See Section 29.4 [Key Bindings], page 269, for more information on key sequences you can bind commands to. See Section 29.4.1 [Keymaps], page 270, for information on creating keymaps.

When we say "*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. The function `next-line` is programmed to move down vertically. *C-n* has this effect *because* it is bound to that function. If you rebind *C-n* to the function `forward-word` then *C-n* will move forward by words instead. Rebinding keys is a common method of customization.

The rest of this manual usually ignores this subtlety to keep things simple. To give the customizer the information needed, we often state the name of the command that 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 uses the variable and behaves differently depending on its setting. 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 29.2 [Variables], page 257.

2.4 XEmacs Pull-down Menus

If you are running XEmacs under X, a menu bar on top of the Emacs frame provides access to pull-down menus of file, edit, and help-related commands. The menus provide convenient shortcuts and an easy interface for novice users. They do not provide additions to the functionality available via key commands; you can still invoke commands from the keyboard as in previous versions of Emacs.

File Perform file and buffer-related operations, such as opening and closing files, saving and printing buffers, as well as exiting Emacs.

Edit Perform standard editing operations, such as cutting, copying, pasting, and killing selected text.

Apps Access to sub-applications implemented within XEmacs, such as the mail reader, the World Wide Web browser, the spell-checker, and the calendar program.

Options Control various options regarding the way XEmacs works, such as controlling which elements of the frame are visible, selecting the fonts to be used for text, specifying whether searches are case-sensitive, etc.

Buffers Present a menu of buffers for selection as well as the option to display a buffer list.

Tools Perform various actions designed to automate software development and similar technical work, such as searching through many files, compiling a program, and comparing or merging two or three files.

Help Access to Emacs Info.

There are two ways of selecting an item from a pull-down menu:

- Select an item in the menu bar by moving the cursor over it and click the left mouse-button. Then move the cursor over the menu item you want to choose and click left again.
- Select an item in the menu bar by moving the cursor over it and click and hold the left mouse-button. With the mouse-button depressed, move the cursor over the menu item you want, then release it to make your selection.

If a command in the pull-down menu is not applicable in a given situation, the command is disabled and its name appears faded. You cannot invoke items that are faded. For example, many commands on the **Edit** menu appear faded until you select text on which they are to operate; after you select a block of text, edit commands are enabled. See Section 9.2 [Mouse Selection], page 66, for information on using the mouse to select text. See Section 10.3 [Using X Selections], page 73, for related information.

There are also `M-x` equivalents for each menu item. To find the equivalent for any left-button menu item, do the following:

1. Type `C-h k` to get the `Describe Key` prompt.
2. Select the menu item and click.

Emacs displays the function associated with the menu item in a separate window, usually together with some documentation.

2.4.1 The File Menu

The **File** menu bar item contains the items **New Frame**, **Open File...**, **Save Buffer**, **Save Buffer As...**, **Revert Buffer**, **Print Buffer**, **Delete Frame**, **Kill Buffer** and **Exit Emacs** on the pull-down menu. If you select a menu item, Emacs executes the equivalent command.

Open File, New Frame...
: Prompts you for a filename and loads that file into a new buffer in a new Emacs frame, that is, a new X window running under the same Emacs process. You can remove the frame using the **Delete Frame** menu item. When you remove the last frame, you exit Emacs and are prompted for confirmation.

Open File...
: Prompts you for a filename and loads that file into a new buffer. **Open File...** is equivalent to the Emacs command `find-file` (`C-x C-f`).

Insert File...
: Prompts you for a filename and inserts the contents of that file into the current buffer. The file associated with the current buffer is not changed by this command. This is equivalent to the Emacs command `insert-file` (`C-x i`).

Save Buffer
: Writes and saves the current Emacs buffer as the latest version of the current visited file. **Save Buffer** is equivalent to the Emacs command `save-buffer` (`C-x C-s`).

Save Buffer As...
: Writes and saves the current Emacs buffer to the filename you specify. **Save Buffer As...** is equivalent to the Emacs command `write-file` (*C-x C-w*).

Revert Buffer
: Restores the last saved version of the file to the current buffer. When you edit a buffer containing a text file, you must save the buffer before your changes become effective. Use **Revert Buffer** if you do not want to keep the changes you have made in the buffer. **Revert Buffer** is equivalent to the Emacs command `revert-file` (*M-x revert-buffer*).

Kill Buffer
: Kills the current buffer, prompting you first if there are unsaved changes. This is roughly equivalent to the Emacs command `kill-buffer` (*C-x k*), except that `kill-buffer` prompts for the name of a buffer to kill.

Print Buffer
: Prints a hardcopy of the current buffer. Equivalent to the Emacs command `print-buffer` (*M-x print-buffer*).

New Frame
: Creates a new Emacs frame displaying the `*scratch*` buffer. This is like the **Open File, New Frame...** menu item, except that it does not prompt for or load a file.

Delete Frame
: Allows you to close all but one of the frames created by **New Frame**. If you created several Emacs frames belonging to the same Emacs process, you can close all but one of them. When you attempt to close the last frame, Emacs informs you that you are attempting to delete the last frame. You have to choose **Exit Emacs** for that.

Split Frame
: Divides the current window on the current frame into two equal-sized windows, both displaying the same buffer. Equivalent to the Emacs command `split-window-vertically` (*C-x 2*).

Un-split (Keep This)
: If the frame is divided into multiple windows, this removes all windows other than the selected one. Equivalent to the Emacs command `delete-other-windows` (*C-x 1*).

Un-split (Keep Others)
: If the frame is divided into multiple windows, this removes the selected window from the frame, giving the space back to one of the other windows. Equivalent to the Emacs command `delete-window` (*C-x 0*).

Exit Emacs
: Shuts down (kills) the Emacs process. Equivalent to the Emacs command `save-buffers-kill-emacs` (*C-x C-c*). Before killing the Emacs process, the system asks which unsaved buffers to save by going through the list of all buffers in that Emacs process.

2.4.2 The Edit Menu

The **Edit** pull-down menu contains the **Undo**, **Cut**, **Copy**, **Paste**, and **Clear** menu items. When you select a menu item, Emacs executes the equivalent command. Most commands on the **Edit** menu work on a block of text, the X selection. They appear faded until you select a block of text (activate a region) with the mouse. See Section 10.3 [Using X Selections], page 73, see Section 10.1 [Killing], page 69, and see Section 10.2 [Yanking], page 71 for more information.

Undo Undoes the previous command. **Undo** is equivalent to the Emacs command `undo` (*C-x u*).

Cut Removes the selected text block from the current buffer, makes it the X clipboard selection, and places it in the kill ring. Before executing this command, you have to select a region using Emacs region selection commands or with the mouse.

Copy Makes a selected text block the X clipboard selection, and places it in the kill ring. You can select text using one of the Emacs region selection commands or by selecting a text region with the mouse.

Paste Inserts the current value of the X clipboard selection in the current buffer. Note that this is not necessarily the same as the Emacs `yank` command, because the Emacs kill ring and the X clipboard selection are not the same thing. You can paste in text you have placed in the clipboard using **Copy** or **Cut**. You can also use **Paste** to insert text that was pasted into the clipboard from other applications.

Clear Removes the selected text block from the current buffer but does not place it in the kill ring or the X clipboard selection.

Start Macro Recording
After selecting this, Emacs will remember every keystroke you type until **End Macro Recording** is selected. This is the same as the Emacs command `start-kbd-macro` (*C-x (*).

End Macro Recording
Selecting this tells emacs to stop remembering your keystrokes. This is the same as the Emacs command `end-kbd-macro` (*C-x)*).

Execute Last Macro
Selecting this item will cause emacs to re-interpret all of the keystrokes which were saved between selections of the **Start Macro Recording** and **End Macro Recording** menu items. This is the same as the Emacs command `call-last-kbd-macro` (*C-x e*).

2.4.3 The Apps Menu

The **Apps** pull-down menu contains the **Read Mail (VM)...**, **Read Mail (MH)...**, **Send Mail...**, **Usenet News**, **Browse the Web**, **Gopher**, **Spell-Check Buffer** and **Emulate VI** menu items, and the **Calendar** and **Games** sub-menus. When you select a menu item, Emacs executes the equivalent command. For some of the menu items, there are sub-menus which you will need to select.

2.4.4 The Options Menu

The **Options** pull-down menu contains the **Read Only**, **Case Sensitive Search**, **Overstrike**, **Auto Delete Selection**, **Teach Extended Commands**, **Syntax Highlighting**, **Paren Highlighting**, **Font**, **Size**, **Weight**, **Buffers Menu Length...**, **Buffers Sub-Menus** and **Save Options** menu items. When you select a menu item, Emacs executes the equivalent command. For some of the menu items, there are sub-menus which you will need to select.

Read Only
Selecting this item will cause the buffer to visit the file in a read-only mode. Changes to the file will not be allowed. This is equivalent to the Emacs command `toggle-read-only` (*C-x C-q*).

Case Sensitive Search
: Selecting this item will cause searches to be case-sensitive. If its not selected then searches will ignore case. This option is local to the buffer.

Overstrike
: After selecting this item, when you type letters they will replace existing text on a one-to-one basis, rather than pushing it to the right. At the end of a line, such characters extend the line. Before a tab, such characters insert until the tab is filled in. This is the same as Emacs command `quoted-insert` (*C-q*).

Auto Delete Selection
: Selecting this item will cause automatic deletion of the selected region. The typed text will replace the selection if the selection is active (i.e. if its highlighted). If the option is not selected then the typed text is just inserted at the point.

Teach Extended Commands
: After you select this item, any time you execute a command with *M-x*which has a shorter keybinding, you will be shown the alternate binding before the command executes.

Syntax Highlighting
: You can customize your `.emacs` file to include the font-lock mode so that when you select this item, the comments will be displayed in one face, strings in another, reserved words in another, and so on. When **Fonts** is selected, different parts of the program will appear in different Fonts. When **Colors** is selected, then the program will be displayed in different colors. Selecting **None** causes the program to appear in just one Font and Color. Selecting **Less** resets the Fonts and Colors to a fast, minimal set of decorations. Selecting **More** resets the Fonts and Colors to a larger set of decorations. For example, if **Less** is selected (which is the default setting) then you might have all comments in green color. Whereas, if **More** is selected then a function name in the comments themselves might appear in a different Color or Font.

Paren Highlighting
: After selecting **Blink** from this item, if you place the cursor on a parenthesis, the matching parenthesis will blink. If you select **Highlight** and place the cursor on a parenthesis, the whole expression of the parenthesis under the cursor will be highlighted. Selecting **None** will turn off the options (regarding **Paren Highlighting**) which you had selected earlier.

Font
: You can select any Font for your program by choosing from one of the available Fonts.

Size
: You can select any size ranging from **2** to **24** by selecting the appropriate option.

Weight
: You can choose either **Bold** or **Medium** for the weight.

Buffers Menu Length...
: Prompts you for the number of buffers to display. Then it will display that number of most recently selected buffers.

Buffers Sub-Menus
: After selection of this item the Buffers menu will contain several commands, as submenus of each buffer line. If this item is unselected, then there are no submenus for each buffer line, the only command available will be selecting that buffer.

Save Options
: Selecting this item will save the current settings of your Options menu to your `.emacs` file.

2.4.5 The Buffers Menu

The **Buffers** menu provides a selection of up to ten buffers and the item **List All Buffers**, which provides a Buffer List. See Section 16.2 [List Buffers], page 126, for more information.

2.4.6 The Tools Menu

The **Tools** pull-down menu contains the **Grep...**, **Compile...**, **Shell Command...**, **Shell Command on Region...**, **Debug(GDB)...** and **Debug(DBX)...** menu items, and the **Compare**, **Merge**, **Apply Patch** and **Tags** sub-menus. When you select a menu item, Emacs executes the equivalent command. For some of the menu items, there are sub-menus which you will need to select.

2.4.7 The Help Menu

The Help Menu gives you access to Emacs Info and provides a menu equivalent for each of the choices you have when using *`C-h`*. See Chapter 8 [Help], page 59, for more information.

The Help menu also gives access to UNIX online manual pages via the **UNIX Manual Page** option.

2.4.8 Customizing XEmacs Menus

You can customize any of the pull-down menus by adding or removing menu items and disabling or enabling existing menu items.

The following functions are available:

`add-menu:` *(menu-path menu-name menu-items &optional before)*
: Add a menu to the menu bar or one of its submenus.

`add-menu-item:` *(menu-path item-name function enabled-p&optional before)*
: Add a menu item to a menu, creating the menu first if necessary.

`delete-menu-item:` *(path)*
: Remove the menu item defined by *path* from the menu hierarchy.

`disable-menu-item:` *(path)*
: Disable the specified menu item.

`enable-menu-item:` *(path)*
: Enable the specified previously disabled menu item.

`relabel-menu-item:` *(path new-name)*
: Change the string of the menu item specified by *path* to *new-name*.

Use the function `add-menu` to add a new menu or submenu. If a menu or submenu of the given name exists already, it is changed.

menu-path identifies the menu under which the new menu should be inserted. It is a list of strings; for example, `("File")` names the top-level **File** menu. `("File" "Foo")` names a hypothetical submenu of **File**. If *menu-path* is `nil`, the menu is added to the menu bar itself.

menu-name is the string naming the menu to be added.

menu-items is a list of menu item descriptions. Each menu item should be a vector of three elements:

- A string, which is the name of the menu item
- A symbol naming a command, or a form to evaluate
- `t` or `nil` to indicate whether the item is selectable

The optional argument *before* is the name of the menu before which the new menu or submenu should be added. If the menu is already present, it is not moved.

The function `add-menu-item` adds a menu item to the specified menu, creating the menu first if necessary. If the named item already exists, the menu remains unchanged.

menu-path identifies the menu into which the new menu item should be inserted. It is a list of strings; for example, `("File")` names the top-level **File** menu. `("File" "Foo")` names a hypothetical submenu of **File**.

item-name is the string naming the menu item to add.

function is the command to invoke when this menu item is selected. If it is a symbol, it is invoked with `call-interactively`, in the same way that functions bound to keys are invoked. If it is a list, the list is simply evaluated.

enabled-p controls whether the item is selectable or not. It should be `t`, `nil`, or a form to evaluate to decide. This form will be evaluated just before the menu is displayed, and the menu item will be selectable if that form returns non-`nil`.

For example, to make the `rename-file` command available from the **File** menu, use the following code:

```
(add-menu-item '("File") "Rename File" 'rename-file t)
```

To add a submenu of file management commands using a **File Management** item, use the following code:

```
(add-menu-item '("File" "File Management") "Copy File" 'copy-file t)
(add-menu-item '("File" "File Management") "Delete File" 'delete-file t)
(add-menu-item '("File" "File Management") "Rename File" 'rename-file t)
```

The optional *before* argument is the name of a menu item before which the new item should be added. If the item is already present, it is not moved.

To remove a specified menu item from the menu hierarchy, use `delete-menu-item`.

path is a list of strings that identify the position of the menu item in the menu hierarchy. `("File" "Save")` means the menu item called **Save** under the top level **File** menu. `("Menu" "Foo" "Item")` means the menu item called **Item** under the **Foo** submenu of **Menu**.

To disable a menu item, use `disable-menu-item`. The disabled menu item is grayed and can no longer be selected. To make the item selectable again, use `enable-menu-item`. `disable-menu-item` and `enable-menu-item` both have the argument *path*.

To change the string of the specified menu item, use `relabel-menu-item`. This function also takes the argument *path*.

new-name is the string to which the menu item will be changed.

3 Entering and Exiting Emacs

The usual way to invoke Emacs is to type `emacs` ⟨RET⟩ at the shell (for XEmacs, type `xemacs` ⟨RET⟩). Emacs clears the screen and then displays an initial advisory message and copyright notice. You can begin typing Emacs commands immediately afterward.

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

Before Emacs reads the first command, you have not had a chance to give a command to specify a file to edit. Since Emacs must always have a current buffer for editing, it presents a buffer, by default, a buffer named '`*scratch*`'. The buffer is in Lisp Interaction mode; you can use it to type Lisp expressions and evaluate them, or you can ignore that capability and simply doodle. You can specify a different major mode for this buffer by setting the variable `initial-major-mode` in your init file. See Section 29.6 [Init File], page 277.

It is possible to give Emacs arguments in the shell command line to specify files to visit, Lisp files to load, and functions to call.

3.1 Exiting Emacs

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

C-z Suspend Emacs (`suspend-emacs`). If used under the X window system, shrink the X window containing the Emacs frame to an icon (see below).

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

If you use XEmacs under the X window system, *C-z* shrinks the X window containing the Emacs frame to an icon. The Emacs process is stopped temporarily, and control is returned to the window manager. If more than one frame is associated with the Emacs process, only the frame from which you used *C-z* is retained. The X windows containing the other Emacs frames are closed.

To activate the "suspended" Emacs, use the appropriate window manager mouse gestures. Usually left-clicking on the icon reactivates and reopens the X window containing the Emacs frame, but the window manager you use determines what exactly happens. To actually kill the Emacs process, use *C-x C-c* or the **Exit Emacs** item on the **File** menu.

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. On these systems, the only way to return to the shell from which Emacs was started (to log out, for example) is to kill Emacs. *C-d* or `exit` are typical commands to exit a subshell.

To kill Emacs, type `C-x C-c` (`save-buffers-kill-emacs`). A two-character key is used for this to make it harder to type. In XEmacs, selecting the **Exit Emacs** option of the **File** menu is an alternate way of issuing the command.

Unless a numeric argument is used, this command first offers to save any modified buffers. If you do not save all buffers, you are asked for reconfirmation with *yes* before killing Emacs, since any changes not saved will be lost. If any subprocesses are still running, `C-x C-c` asks you to confirm killing them, since killing Emacs kills the subprocesses simultaneously.

In most programs running on Unix, certain characters may instantly suspend or kill the program. (In Berkeley Unix these characters are normally `C-z` and `C-c`.) *This Unix feature is turned off while you are in Emacs.* The meanings of `C-z` and `C-x C-c` as keys in Emacs were inspired by the standard Berkeley Unix meanings of `C-z` and `C-c`, but that is their only relationship with Unix. You could customize these keys to do anything (see Section 29.4.1 [Keymaps], page 270).

3.2 Command Line Switches and Arguments

XEmacs supports command line arguments you can use to request various actions when invoking Emacs. The commands are for compatibility with other editors and for sophisticated activities. If you are using XEmacs under the X window system, you can also use a number of standard Xt command line arguments. Command line arguments are not usually needed for editing with Emacs; new users can skip this section.

Many editors are designed to be started afresh each time you want to edit. You start the editor to edit one file; then exit the editor. The next time you want to edit either another file or the same one, you start the editor again. Under these circumstances, it makes sense to use a command line argument to say which file to edit.

The recommended way to use XEmacs is to start it only once, just after you log in, and do all your editing in the same Emacs process. Each time you want to edit a file, you visit it using the existing Emacs. Emacs creates a new buffer for each file, and (unless you kill some of the buffers) Emacs eventually has many files in it ready for editing. Usually you do not kill the Emacs process until you are about to log out. Since you usually read files by typing commands to Emacs, command line arguments for specifying a file when Emacs is started are seldom needed.

Emacs accepts command-line arguments that specify files to visit, functions to call, and other activities and operating modes. If you are running XEmacs under the X window system, a number of standard Xt command line arguments are available as well.

The following subsections list:

- Command line arguments that you can always use
- Command line arguments that have to appear at the beginning of the argument list
- Command line arguments that are only relevant if you are running XEmacs under X

3.2.1 Command Line Arguments for Any Position

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

Here are the arguments allowed:

'*file*'
: Visit *file* using `find-file`. See Section 15.2 [Visiting], page 102.

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

'`-load` *file*'
'`-l` *file*'
: Load a file *file* of Lisp code with the function `load`. See Section 23.3 [Lisp Libraries], page 191.

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

'`-eval` *function*'
: Interpret the next argument as a Lisp expression, and evaluate it. You must be very careful of the shell quoting here.

'`-insert` *file*'
'`-i` *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 123.

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

'`-version`'
'`-V`'
: Prints version information. This implies '`-batch`'.

```
% xemacs -version
XEmacs 19.13 of Mon Aug 21 1995 on willow (usg-unix-v) [formerly Lucid Emacs]
```

'`-help`'
: Prints a summary of command-line options and then exits.

3.2.2 Command Line Arguments (Beginning of Line Only)

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

'`-t` *file*'
: Use *file* instead of the terminal for input and output. This implies the '`-nw`' option, documented below.

'`-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 `stderr` only what would normally be printed in the echo area under program control.

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

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

'-nw' Start up XEmacs in TTY mode (using the TTY XEmacs was started from), rather than trying to connect to an X display. Note that this happens automatically if the 'DISPLAY' environment variable is not set.

'-debug-init'
: Enter the debugger if an error in the init file occurs.

'-debug-paths'
: Displays information on how XEmacs constructs the various paths into its hierarchy on startup. (See also see Section 3.3 [Startup Paths], page 38.)

'-unmapped'
: Do not map the initial frame. This is useful if you want to start up XEmacs as a server (e.g. for gnuserv screens or external client widgets).

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

'-no-site-file'
: Do not load the site-specific init file 'lisp/site-start.el'.

'-no-autoloads'
: Do not load global symbol files ('auto-autoloads') at startup. This implies '-vanilla'.

'-no-early-packages'
: Do not process early packages. (For more information on startup issues concerning the package system, See Section 3.3 [Startup Paths], page 38.)

'-vanilla'
: This is equivalent to '-q -no-site-file -no-early-packages'.

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

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

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

```
xemacs *.c
```

passes each `.c` file as a separate argument to Emacs, so that Emacs visits each file (see Section 15.2 [Visiting], page 102).

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

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

Here Emacs is told to visit 'foo.c', load 'hack-c-program.el' (which makes changes in the visited file), save 'foo.c' (note that `save-buffer` is the function that *C-x C-s* is bound to), and then exit to the shell from which the command was executed. '-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.

3.2.3 Command Line Arguments (for XEmacs Under X)

If you are running XEmacs under X, a number of options are available to control color, border, and window title and icon name:

'`-title` *title*'
'`-wn` *title*'
'`-T` *title*'
: Use *title* as the window title. This sets the `frame-title-format` variable, which controls the title of the X window corresponding to the selected frame. This is the same format as `mode-line-format`.

'`-iconname` *title*'
'`-in` *title*'
: Use *title* as the icon name. This sets the `frame-icon-title-format` variable, which controls the title of the icon corresponding to the selected frame.

'`-mc` *color*'
: Use *color* as the mouse color.

'`-cr` *color*'
: Use *color* as the text-cursor foreground color.

'`-private`'
: Install a private colormap for XEmacs.

In addition, XEmacs allows you to use a number of standard Xt command line arguments.

'`-background` *color*'
'`-bg` *color*'
: Use *color* as the background color.

'`-bordercolor` *color*'
'`-bd` *color*'
: Use *color* as the border color.

'`-borderwidth` *width*'
'`-bw` *width*'
: Use *width* as the border width.

'`-display` *display*'
'`-d` *display*'
: When running under the X window system, create the window containing the Emacs frame on the display named *display*.

'`-foreground` *color*'
'`-fg` *color*'
: Use *color* as the foreground color.

'`-font` *name*'
'`-fn` *name*'
: Use *name* as the default font.

'`-geometry` *spec*'
'`-geom` *spec*'
'`-g` *spec*'
: Use the geometry (window size and/or position) specified by *spec*.

'`-iconic`'
: Start up iconified.

'`-rv`'
: Bring up Emacs in reverse video.

'-name *name*'
: Use the resource manager resources specified by *name*. The default is to use the name of the program (argv[0]) as the resource manager name.

'-xrm'
: Read something into the resource database for this invocation of Emacs only.

3.3 How XEmacs finds Directories and Files

XEmacs deals with a multitude of files during operation. These files are spread over many directories, and XEmacs determines the location of most of these directories at startup and organizes them into various paths. (A *path*, for the purposes of this section, is simply a list of directories which XEmacs searches successively in order to locate a file.)

3.3.1 XEmacs Directory Hierarchies

Many of the files XEmacs looks for are located within the XEmacs installation itself. However, there are several views of what actually constitutes the "XEmacs installation": XEmacs may be run from the compilation directory, it may be installed into arbitrary directories, spread over several directories unrelated to each other. Moreover, it may subsequently moved to a different place. (This last case is not as uncommon as it sounds. Binary kits work this way.) Consequently, XEmacs has quite complex procedures in place to find directories, no matter where they may be hidden.

XEmacs will always respect directory options passed to configure. However, if it cannot locate a directory at the configured place, it will initiate a search for the directory in any of a number of *hierachies* rooted under a directory which XEmacs assumes contain parts of the XEmacs installation; it may locate several such hierarchies and search across them. (Typically, there are just one or two hierarchies: the hierarchy where XEmacs was or will be installed, and the one where it is being built.) Such a directory containing a hierarchy is called a *root*. Whenever this section refers to a directory using the shorthand <root>, it means that XEmacs searches for it under all hierarchies under all hierarchies XEmacs was able to scrounge up. In a running XEmacs, the hierarchy roots are stored in the variable emacs-roots.

3.3.2 Package Hierarchies

Many relevant directories and files XEmacs uses are actually not part of the core installation. They are part of any of the many packages usually installed on top of an XEmacs installation. (See Section 23.8 [Packages], page 197.) Hence, they play a prominent role in the various paths XEmacs sets up.

XEmacs locates packages in any of a number of package hierarchies. Package hierarchies fall into three groups: *early*, *late*, and *last*, according to the relative location at which they show up in the various XEmacs paths. Early package hierarchies are at the very front, late ones somewhere in the middle, and last hierarchies are (you guessed it) last.

By default, XEmacs expects an early package hierarchy in the a subdirectory '.xemacs' of the user's home directory.

Moreover, XEmacs expects late hierarchies in the subdirectories 'site-packages', 'mule-packages', and 'xemacs-packages' (in that order) of the '<root>/lib/xemacs' subdirectory of one of the installation hierarchies. (If you run in-place, these are direct subdirectories of the build directory.) Furthermore, XEmacs will also search these subdirectories in the '`<root>/lib/xemacs-<VERSION>`' subdirectory and prefer directories found there.

By default, XEmacs does not have a pre-configured last package hierarchy. Last hierarchies are primarily for using package hierarchies of outdated versions of XEmacs as a fallback option. For example, it is possible to run XEmacs 21.0 with the 20.4 package hierarchy as a last hierarchy.

It is possible to specify at configure-time the location of the various package hierarchies with the `--package-path` option to configure. The early, late, and last components of the package path are separated by double instead of single colons. If three components are present, they are locate the early, late, and last package hierarchies respectively. If two components are present, they locate the early and late hierarchies. If only one component is present, it locates the late hierarchy. At run time, the package path may also be specified via the `PACKAGEPATH` environment variable.

An XEmacs package is laid out just like a normal installed XEmacs lisp directory. It may have '`lisp`', '`etc`', '`info`', and '`lib-src`' subdirectories. XEmacs adds these at appropriate places within the various system-wide paths.

There may be any number of package hierarchy directories.

3.3.3 Directories and Paths

Here is a list of the various directories and paths XEmacs tries to locate during startup. XEmacs distinguishes between directories and paths specific to *version*, *site*, and *architecture* when looking for them.

`version-specific`
: directories are specific to the version of XEmacs they belong to and typically reside under '`<root>/lib/xemacs-<VERSION>`'.

`site-specific`
: directories are independent of the version of XEmacs they belong to and typically reside under '`<root>/lib/xemacs`'

`architecture-specific`
: directories are specific both to the version of XEmacs and the architecture it runs on and typically reside under '`<root>/lib/xemacs-<VERSION>/<ARCHITECTURE>`'.

During installation, all of these directories may also reside directly under '`<root>`', because that is where they are in the XEmacs tarball.

If XEmacs runs with the `-debug-paths` option (see Section 3.2 [Command Switches], page 34), it will print the values of these variables, hopefully aiding in debugging any problems which come up.

`lisp-directory`
: Contains the version-specific location of the Lisp files that come with the core distribution of XEmacs. XEmacs will search it recursively to a depth of 1 when setting up `load-path`.

`load-path`
: Is where XEmacs searches for XEmacs Lisp files with commands like `load-library`. It contains the package lisp directories (see further down) and the version-specific core Lisp directories. If the environment variable `EMACSLOADPATH` is set at startup, its directories are prepended to `load-path`.

`Info-directory-list`
: Contains the location of info files. (See ⟨undefined⟩ [(info)], page ⟨undefined⟩.) It contains the package info directories and the version-specific core documentation. Moreover, XEmacs will add '`/usr/info`', '`/usr/local/info`' as well as the directories of the environment variable `INFOPATH` to `Info-directory-list`.

`lock-directory`
`superlock-file`
: Are the site-specific locations of the lock directory and the superlock file, respectively. The `lock-directory` variable may also be initialized from the `EMACSLOCKDIR` environment variable.

`exec-directory`
: Is the directory of architecture-dependent files that come with XEmacs, especially executable programs intended for XEmacs to invoke.

`exec-path`
: Is the path for executables which XEmacs may want to start. It contains the package executable paths as well as `exec-directory`, and the directories of the environment variables `PATH` and `EMACSPATH`.

`doc-directory`
: Is the directory containing the architecture-specific '`DOC`' file that contains documentation for XEmacs' commands.

`data-directory`
: Is the version-specific directory that contains core data files XEmacs uses. It may be initialized from the `EMACSDATA` environment variable.

`data-directory-list`
: Is the path where XEmacs looks for data files. It contains package data directories as well as `data-directory`.

4 Basic Editing Commands

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

4.1 Inserting Text

To insert printing characters into the text you are editing, just type them. This inserts the characters into the buffer at the cursor (that is, at *point*; see Section 1.1 [Point], page 14). The cursor moves forward. Any characters after the cursor move forward too. If the text in the buffer is '`FOOBAR`', with the cursor before the '`B`', and you type *`XX`*, the result is '`FOOXXBAR`', with the cursor still before the '`B`'.

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

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

Emacs automatically splits lines when they become too long, if you turn on a special mode called *Auto Fill* mode. See Section 21.6 [Filling], page 159, for information on using 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. There are two ways to use *`C-q`*:

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

A numeric argument to *`C-q`* specifies how many copies of the quoted character should be inserted (see Section 4.9 [Arguments], page 46).

If you prefer to have text characters replace (overwrite) existing text instead of moving it to the right, you can enable Overwrite mode, a minor mode. See Section 29.1 [Minor Modes], page 257.

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 14). Here are a few of the available commands.

NOTE: Many of the following commands have two versions, one that uses the function keys (e.g. ⟨LEFT⟩ or ⟨END⟩) and one that doesn't. The former versions may only be available on X terminals (i.e. not on TTY's), but the latter are available on all terminals.

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

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

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

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

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

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

C-n
DOWN 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
UP Move up one line, vertically (`previous-line`).

C-v
PGDN Move down one page, vertically (`scroll-up`).

M-v
PGUP Move up one page, vertically (`scroll-down`).

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

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

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

M-<
C-HOME 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 46, for more information on numeric arguments.

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

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

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

C-x C-n
: Use the current column of point as the *semi-permanent 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, *C-n* and *C-p* move to the end of the line when at the end of the starting line. By default, `track-eol` is `nil`.

4.3 Erasing Text

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

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

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

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

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

In contrast to the ⟨DEL⟩ key, which deletes the character before the cursor, *Control-d* deletes the character after the cursor, causing the rest of the text on the line to shift left. If *Control-d* is typed at the end of a line, that line and the next line are joined.

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

See Section 10.1 [Killing], page 69, for more flexible ways of killing text.

4.4 Files

The commands above are sufficient for creating and altering text in an Emacs buffer. 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 and which you can retrieve 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`'. To begin editing this file from Emacs, type:

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

The file name is given as an *argument* to the command *C-x C-f* (`find-file`). The command uses the *minibuffer* to read the argument. You have to type ⟨RET⟩ to terminate the argument (see Chapter 6 [Minibuffer], page 51).

You can also use the **Open...** menu item from the **File** menu, then type the name of the file to the prompt.

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

To create a file, visit the file with *C-x C-f* as if it already existed or choose **Open...** from the **File** menu and provide the name for the new file in the minibuffer. Emacs will create an empty buffer in which you can insert the text you want to put in the file. When you save the buffer with *C-x C-s*, or by choosing **Save Buffer** from the **File** menu, the file is created.

To learn more about using files, see Chapter 15 [Files], page 101.

4.5 Help

If you forget what a key does, you can use the Help character (*C-h*) to find out: Type *C-h k* followed by the key you want to know about. For example, *C-h k C-n* tells you 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 59.

4.6 Blank Lines

Here are special commands and techniques for entering and removing 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 just type the new line of text, followed by ⟨RET⟩. If you prefer to create a blank line first and then insert the desired text, use the key *C-o* (`open-line`), which inserts a newline after point but leaves point in front of the newline. Then type the text into the new line. *C-o F O O* has the same effect as *F O O* ⟨RET⟩, except for the final location of point.

To create several blank lines, type *C-o* several times, or give *C-o* an argument indicating how many blank lines to create. See Section 4.9 [Arguments], page 46, for more information.

If you have many blank lines in a row and want to get rid of them, use *C-x C-o* (`delete-blank-lines`). If point is on a blank line which is adjacent to at least one other blank line, *C-x C-o* deletes all but one of the blank lines. If point is on a blank line with no other adjacent blank line, the sole blank line is deleted. If point is on a non-blank line, *C-x C-o* deletes any blank lines following that non-blank line.

4.7 Continuation Lines

If you add too many characters to one line without breaking with a ⟨RET⟩, the line grows to occupy two (or more) screen lines, with a curved arrow at the extreme right margin of all but the last line. The curved arrow indicates 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 frame. You can use Auto Fill mode (see Section 21.6 [Filling], page 159) to have Emacs insert newlines automatically when a line gets too long.

Instead of continuation, long lines can be displayed by *truncation*. This means that all the characters that do not fit in the width of the frame or window do not appear at all. They remain in the buffer, temporarily invisible. Three diagonal dots in the last column (instead of the curved arrow inform you that truncation is in effect.

To turn off continuation for a particular buffer, set the variable `truncate-lines` to non-`nil` in that buffer. Truncation instead of continuation also happens whenever horizontal scrolling is in use, and optionally whenever side-by-side windows are in use (see Chapter 17 [Windows], page 131). 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 29.2.4 [Locals], page 264.

4.8 Cursor Position Information

If you are accustomed to other display editors, you may be surprised that Emacs does not always display the page number or line number of point in the mode line. In Emacs, this information is only rarely needed, and a number of commands are available to compute and print it. Since text is stored in a way that makes it difficult to compute the information, it is not displayed all the time.

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

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

M-=
: Print number of lines and characters 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 several commands for printing line numbers:

- *M-x what-line* counts lines from the beginning of the file and prints the line number point is on. The first line of the file is line number 1. You can use these numbers as arguments to *M-x goto-line*.

- *M-x what-page* counts pages from the beginning of the file, and counts lines within the page, printing both of them. See Section 21.5 [Pages], page 158, for the command *C-x l*, which counts the lines in the current page.
- *M-=* (`count-lines-region`) prints the number of lines in the region (see Chapter 9 [Mark], page 63).

The command *C-x =* (`what-cursor-position`) provides information about point and about the column the cursor is in. It prints a line in the echo area that looks like this:

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

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

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

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

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

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

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

where the two extra numbers give the smallest and largest character position that point is allowed to assume. The characters between those two positions are the visible ones. See Section 28.3 [Narrowing], page 252.

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

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

4.9 Numeric Arguments

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

If your keyboard has a ⟨META⟩ key (labelled with a diamond on Sun-type keyboards and labelled 'Alt' on some other keyboards), the easiest way to specify a numeric argument is to type digits and/or a minus sign while holding down the ⟨META⟩ key. For example,

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

moves 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 frame. Other useful combinations are *C-u C-n*, *C-u C-u C-n* (move down a good fraction of a frame), *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 21.6 [Filling], page 159, 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 non-blank line, just like *C-k* with an argument of one. (See Section 10.1 [Killing], page 69, 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 to make the individual command more convenient to use.

5 Undoing Changes

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

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

C-_ The same.

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

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

Any command other than an undo command breaks the sequence of undo commands. Starting at this moment, the previous undo commands are considered ordinary changes that can themselves be undone. Thus, you can redo changes you have undone by typing *C-f* or any other command that 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. When that happens, all the modifications you made have been canceled. If you do not remember whether you changed the buffer deliberately, type *C-_* once. When you see Emacs undo the last change you made, you probably remember why you made it. If the change 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, the buffer contents is the same as it was 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. Minibuffers, help buffers, and documentation buffers also don't record undo information.

Emacs can remember at most 8000 or so characters of deleted or modified text in any one buffer for reinsertion by the undo command. There is also a limit on the number of individual insert, delete, or change actions that Emacs can remember.

There are two keys to run the `undo` command, *C-x u* and *C-_*, because 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

Emacs commands use the *minibuffer* 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. To edit the argument in the minibuffer, you can use Emacs editing commands.

When the minibuffer is in use, it appears in the echo area, and the cursor moves there. The beginning of the minibuffer line displays a *prompt* indicating what kind of input you should supply and how it will be used. The prompt is often derived from the name of the command 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 is used as the argument value if you enter an empty argument (e.g., by just typing ⟨RET⟩). For example, commands that read buffer names always show a default, which is the name of the buffer that will be used if you type just ⟨RET⟩.

The simplest way to give a minibuffer argument is to type the text you want, terminated by ⟨RET⟩ to exit the minibuffer. To get out of the minibuffer and cancel the command that it was for, type `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 you use a command in the minibuffer whose purpose is to print a message in the echo area (for example `C-x =`) the message is displayed 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 informs you in which directory the file will be looked for if you do not specify a different one. For example, the minibuffer might start out with:

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

where '`Find File: `' is the prompt. Typing `buffer.c` specifies the file '`/u2/emacs/src/buffer.c`'. To find files in nearby directories, use '`..`'; thus, if you type `../lisp/simple.el`, the file that you visit will be the one named '`/u2/emacs/lisp/simple.el`'. Alternatively, you can use `M-`⟨DEL⟩ to kill directory names you don't want (see Section 21.2 [Words], page 156).

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

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

Two slashes in a row are not normally meaningful in Unix file names, but they are allowed in XEmacs. 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, you must use *C-o* or *C-q* ⟨LFD⟩ to insert a newline into the minibuffer. (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. The minibuffer 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 131.

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

If you are in the minibuffer and issue a command that displays help text in another window, that window will be scrolled if you type *M-C-v* while in the minibuffer until you exit the minibuffer. This feature is helpful if a completing minibuffer gives you a long list of possible completions.

If the variable `minibuffer-confirm-incomplete` is `t`, you are asked for confirmation if there is no known completion for the text you typed. For example, if you attempted to visit a non-existent file, the minibuffer might read:

```
Find File:chocolate_bar.c [no completions, confirm]
```

If you press *Return* again, that confirms the filename. Otherwise, you can continue editing it.

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

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

6.3 Completion

When appropriate, the minibuffer provides a *completion* facility. You type the beginning of an argument and one of the completion keys, and Emacs visibly fills in the rest, depending on what you have already typed.

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

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

Case is normally significant in completion because it is significant in most of the names that you can complete (buffer names, file names, and command names). Thus, '`fo`' will not complete to '`Foo`'. When you are completing a name in which case does not matter, case may be ignored for completion's sake if specified by program.

When a completion list is displayed, the completions will highlight as you move the mouse over them. Clicking the middle mouse button on any highlighted completion will "select" it just as if you had typed it in and hit ⟨RET⟩.

6.3.1 A Completion Example

Consider the following example. If you type *Meta-x au* ⟨TAB⟩, ⟨TAB⟩ looks for alternatives (in this case, command names) that start with '`au`'. There are only two commands: `auto-fill-mode` and `auto-save-mode`. They are the same as far as `auto-`, so the '`au`' in the minibuffer changes to '`auto-`'.

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

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

6.3.2 Completion Commands

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

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

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

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

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

⟨button2⟩ Select the highlighted text under the mouse as a minibuffer response. When the minibuffer is being used to prompt the user for a completion, any valid completions which are visible on the screen will be highlighted when the mouse moves over them. Clicking ⟨button2⟩ will select the highlighted completion and exit the minibuffer. (`minibuf-select-highlighted-completion`).

⟨SPC⟩ completes in a way that is similar to ⟨TAB⟩, but it 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-`'. The result is '`auto-fill-`'. Another ⟨SPC⟩ at this point completes all the way to '`auto-fill-mode`'. ⟨SPC⟩ in the minibuffer runs the function `minibuffer-complete-word` when completion is available.

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

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

  Cautious completion is used for reading file names for files that must already exist.
- *Permissive* completion is used when any string 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. Typing *?* explicitly requests such a list. The list of completions counts as help text, so *C-M-v* typed in the minibuffer scrolls the list.

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

If a completion command finds the next character is undetermined, it 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.

If the variable `minibuffer-confirm-incomplete` is set to `t`, then in contexts where `completing-read` allows answers that are not valid completions, an extra ⟨RET⟩ must be typed to confirm the response. This is helpful for catching typos.

6.4 Repeating Minibuffer Commands

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

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

M-p
: Within *C-x* ⟨ESC⟩, move to previous recorded command (`previous-history-element`).

M-n
: Within *C-x* ⟨ESC⟩, move to the next (more recent) recorded command (`next-history-element`).

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

C-x ⟨ESC⟩ is used to re-execute a recent command that used the minibuffer. With no argument, it repeats the last command. A numeric argument specifies which command to repeat; 1 means the last one, and larger numbers specify earlier commands.

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

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

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

M-n and *M-p* are specially defined within *C-x* ⟨ESC⟩ to run the commands `previous-history-element` and `next-history-element`.

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

7 Running Commands by Name

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

A command name consists, by convention, 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 results in more characters to type. You can run any command by name, even if it can be run by keys as well.

To run a command by name, start with *M-x*, then type the command name, and finish with ⟨RET⟩. *M-x* uses the minibuffer to read the command name. ⟨RET⟩ exits the minibuffer and runs the command.

Emacs uses the minibuffer for reading input for many different purposes; on this occasion, the string '`M-x`' is displayed at the beginning of the minibuffer as a *prompt* to remind you that your input should be the name of a command to be run. See Chapter 6 [Minibuffer], page 51, for full information on the features of the minibuffer.

You can use completion to enter a command name. For example, to invoke the command `forward-char`, type:

```
M-x forward-char ⟨RET⟩
```

or

```
M-x fo ⟨TAB⟩ c ⟨RET⟩
```

After you type in `M-x fo TAB` emacs will give you a possible list of completions from which you can choose. Note that `forward-char` is the same command that you invoke with the key *C-f*. You can call any command (interactively callable function) defined in Emacs by its name using *M-x* regardless of whether or not any keys are bound to it.

If you type *C-g* while Emacs reads the command name, you cancel the *M-x* command and get out of the minibuffer, ending up at top level.

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

You can use the command `M-x interactive` to specify a way of parsing arguments for interactive use of a function. For example, write:

```
(defun foo (arg) "Doc string" (interactive "p") ...use arg...)
```

to make `arg` be the prefix argument when `foo` is called as a command. The call to `interactive` is actually a declaration rather than a function; it tells `call-interactively` how to read arguments to pass to the function. When actually called, `interactive` returns `nil`.

The argument of *interactive* is usually a string containing a code letter followed by a prompt. Some code letters do not use I/O to get the argument and do not need prompts. To prompt for multiple arguments, you must provide a code letter, its prompt, a newline, and another code letter, and so forth. If the argument is not a string, it is evaluated to get a list of arguments to pass to the function. If you do not provide an argument to `interactive`, no arguments are passed when calling interactively.

Available code letters are:

`a` Function name: symbol with a function definition

`b` Name of existing buffer

`B` Name of buffer, possibly nonexistent

`c` Character

`C` Command name: symbol with interactive function definition

`d` Value of point as number (does not do I/O)

`D` Directory name

`e` Last mouse event

`f` Existing file name

`F` Possibly nonexistent file name

`k` Key sequence (string)

`m` Value of mark as number (does not do I/O)

`n` Number read using minibuffer

`N` Prefix arg converted to number, or if none, do like code `n`

`p` Prefix arg converted to number (does not do I/O)

`P` Prefix arg in raw form (does not do I/O)

`r` Region: point and mark as two numeric arguments, smallest first (does not do I/O)

`s` Any string

`S` Any symbol

`v` Variable name: symbol that is `user-variable-p`

`x` Lisp expression read but not evaluated

`X` Lisp expression read and evaluated

In addition, if the string begins with '`*`', an error is signaled if the buffer is read-only. This happens before reading any arguments. If the string begins with '`@`', the window the mouse is over is selected before anything else is done. You may use both '`@`' and '`*`'; they are processed in the order that they appear.

Normally, when describing a command that is run by name, we omit the ⟨RET⟩ that is needed to terminate the name. Thus we may refer to *M-x auto-fill-mode* rather than *M-x auto-fill-mode* ⟨RET⟩. We mention the ⟨RET⟩ only when it is necessary to emphasize its presence, for example, when describing a sequence of input that contains a command name and arguments that follow it.

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

8 Help

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

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

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

You should type one of those characters.

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

Here is a summary of the defined help commands.

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

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

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

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

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

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

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

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

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

C-h p
: Display a table of all mouse bindings currently in effect now, with local bindings of the current major mode first, followed by all global bindings (`describe-pointer`).

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

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

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

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

`M-x apropos` *regexp*
Show all symbols whose names contain matches for *regexp*.

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 the name of the command that *key* is bound to in the echo area. For example, `C-h c C-f` prints '`forward-char`'. Since command names are chosen to describe what the command does, using this option is a good way to get a somewhat cryptic description of what *key* does.

`C-h k` *key* is similar to `C-h c` but gives more information. It displays the documentation string of the function *key* is bound to as well as its name. *key* is a string or vector of events. When called interactively, *key* may also be a menu selection. This information does not usually fit into the echo area, so a window is used for the display.

8.2 Help by Command or Variable Name

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

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

displays the documentation for `auto-fill-mode`. Using `C-h f` is the only way to see the documentation of a command that is not bound to any key, that is, a command you would normally call using `M-x`. If the variable `describe-function-show-arglist` is `t`, `describe-function` shows its arglist if the *function* is not an autoload function.

`C-h f` is also useful for Lisp functions you are planning to use in a Lisp program. For example, if you have just written the code `(make-vector len)` and want to make sure 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.

If you type ⟨RET⟩, leaving the minibuffer empty, `C-h f` by default describes the function called by the innermost Lisp expression in the buffer around point, *provided* that 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 starting 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 what you wanted to know, just type `C-g` to cancel the `C-h f` command and continue editing.

C-h w command ⟨RET⟩ (`where-s`) tells you what keys are bound to *command*. It prints a list of the keys in the echo area. Alternatively, it informs you that a command is not bound to any keys, which implies that you must use *M-x* to call the command.

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, if that is the name of a known Lisp variable. See Section 29.2 [Variables], page 257.

8.3 Apropos

C-h `a` Show only symbols that are names of commands (`command-apropos`).

M-x `apropos` *regexp*
: Show all symbols whose names comtain matches for *regexp*.

It is possible to ask a question like, "What are the commands for working with files?" To do this, type *C-h* `a` *file* ⟨RET⟩, which displays a list of all command names that contain '`file`', such as `copy-file`, `find-file`, and so on. With each command name a brief description of its use and information on the keys you can use to invoke it is displayed. For example, you would be informed that you can invoke `find-file` by typing *C-x C-f*. The `a` in *C-h* `a` stands for 'Apropos'; *C-h* `a` runs the Lisp function `command-apropos`.

Because *C-h* `a` looks only for functions whose names contain the string 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 commands, don't give up. Try just *kill*, or just *backwards*, or just *back*. Be persistent. Pretend you are playing Adventure. Also note that you can use a regular expression as the argument (see Section 13.5 [Regexps], page 89).

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 standard Emacs commands. By giving you a feeling for the naming conventions, this set of arguments can also help you develop a technique for picking `apropos` strings.

> char, line, word, sentence, paragraph, region, page, sexp, list, defun, 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`.

8.4 Other Help Commands

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

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

Emacs has several major modes. Each mode 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 information about the current Emacs mode that is not covered by *C-h m*. *C-h b* displays a list of all key bindings currently in effect, with the local bindings of the current major mode first, followed by the global bindings (see Section 29.4 [Key Bindings], page 269). *C-h s* displays the contents of the syntax table with explanations of each character's syntax (see Section 29.5 [Syntax], page 274).

The other *C-h* options display various files of useful information. *C-h C-w* (`describe-no-warranty`) displays details on the complete absence of warranty for XEmacs. *C-h n* (`view-emacs-news`) displays the file '`emacs/etc/NEWS`', which contains documentation on Emacs changes arranged chronologically. *C-h t* (`help-with-tutorial`) displays the learn-by-doing Emacs tutorial. *C-h C-c* (`describe-copying`) displays the file '`emacs/etc/COPYING`', which tells you the conditions you must obey in distributing copies of Emacs. *C-h C-d* (`describe-distribution`) displays another file named '`emacs/etc/DISTRIB`', which tells you how you can order a copy of the latest version of Emacs.

9 Selecting Text

Many Emacs commands operate on an arbitrary contiguous part of the current buffer. You can select text in two ways:

- You use special keys to select text by defining a region between point and the mark.
- If you are running XEmacs under X, you can also select text with the mouse.

9.1 The Mark and the Region

To specify the text for a command to operate on, set *the mark* at one end of it, and move point to the other end. The text between point and the mark is called *the region*. You can move point or the mark to adjust the boundaries of the region. It doesn't matter which one is set first chronologically, or which one comes earlier in the text.

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

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

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

9.1.1 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`).

`C-<` Pushes a mark at the beginning of the buffer.

`C->` Pushes a mark at the end of the buffer.

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

The most common way to set the mark is with the `C-SPC` command (`set-mark-command`). This command sets the mark where point is. You can then move point away, leaving the mark behind. It is actually incorrect to speak of the character `C-SPC`; there is no such character. When you type `SPC` while holding down `CTRL`, you get the character `C-@` on most terminals. This character 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 should think of this character as `C-SPC`.

Since terminals have only one cursor, Emacs cannot show you where the mark is located. Most people use the mark soon after they set it, before they forget where it is. But you can see where the mark is with the command `C-x C-x` (`exchange-point-and-mark`) which puts the mark where point was and point where the mark was. The extent of the region is unchanged, but the cursor and point are now at the previous location of the mark.

Another way to set the mark is to push the mark to the beginning of a buffer while leaving point at its original location. If you supply an argument to `C-<` (`mark-beginning-of-buffer`), the mark is pushed $n/10$ of the way from the true beginning of the buffer. You can also set the mark at the end of a buffer with `C->` (`mark-end-of-buffer`). It pushes the mark to the end of the buffer, leaving point alone. Supplying an argument to the command pushes the mark $n/10$ of the way from the true end of the buffer.

If you are using XEmacs under the X window system, you can set the variable `zmacs-regions` to `t`. This makes the current region (defined by point and mark) highlight and makes it available as the X clipboard selection, which means you can use the menu bar items on it. See Section 10.3.4 [Active Regions], page 75, for more information.

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

9.1.2 Operating on the Region

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

- Kill it with `C-w` (see Section 10.1 [Killing], page 69).
- Save it in a register with `C-x r s` (see Chapter 11 [Registers], page 79).
- Save it in a buffer or a file (see Section 10.4 [Accumulating Text], page 76).
- Convert case with `C-x C-l` or `C-x C-u`
  (see Section 21.7 [Case], page 162).
- Evaluate it as Lisp code with `M-x eval-region` (see Section 23.4 [Lisp Eval], page 193).
- Fill it as text with `M-q` (see Section 21.6 [Filling], page 159).
- Print hardcopy with `M-x print-region` (see Section 28.4 [Hardcopy], page 253).
- Indent it with `C-x TAB` or `C-M-\` (see Chapter 20 [Indentation], page 145).

9.1.3 Commands to Mark Textual Objects

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

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

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

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

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

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

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

M-@ (`mark-word`) puts the mark at the end of the next word, while *C-M-@* (`mark-sexp`) puts it at the end of the next Lisp expression. These characters sometimes save you some typing.

A number of commands are available that set both point and mark and thus delimit an object in the buffer. *M-h* (`mark-paragraph`) moves point to the beginning of the paragraph that surrounds or follows point, and puts the mark at the end of that paragraph (see Section 21.4 [Paragraphs], page 158). You can then indent, case-convert, or kill the whole paragraph. In the same fashion, *C-M-h* (`mark-defun`) puts point before and the mark after the current or following defun (see Section 22.3 [Defuns], page 166). *C-x C-p* (`mark-page`) puts point before the current page (or the next or previous, depending on the argument), and mark at the end (see Section 21.5 [Pages], page 158). The mark goes after the terminating page delimiter (to include it), while point goes after the preceding page delimiter (to exclude it). Finally, *C-x h* (`mark-whole-buffer`) sets up the entire buffer as the region by putting point at the beginning and the mark at the end.

9.1.4 The Mark Ring

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

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

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

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

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

9.2 Selecting Text with the Mouse

If you are using XEmacs under X, you can use the mouse pointer to select text. (The normal mouse pointer is an I-beam, the same pointer that `xterm` uses.)

The glyph variable `text-pointer-glyph` controls the shape of the mouse pointer when over text. You can also control the shape of the mouse pointer when over nontext using `nontext-pointer-glyph`, and the shape of the mouse pointer when over the modeline using `modeline-pointer-glyph`. (Remember, you should use `set-glyph-image`, not `setq`, to set one of these variables.)

If you want to get fancy, you can set the foreground and background colors of the mouse pointer by setting the `pointer` face.

There are two ways to select a region of text with the mouse:

To select a word in text, double-click with the left mouse button while the mouse cursor is over the word. The word is highlighted when selected. On monochrome monitors, a stippled background indicates that a region of text has been highlighted. On color monitors, a color background indicates highlighted text. You can triple-click to select whole lines.

To select an arbitrary region of text:

1. Move the mouse cursor over the character at the beginning of the region of text you want to select.
2. Press and hold the left mouse button.
3. While holding the left mouse button down, drag the cursor to the character at the end of the region of text you want to select.
4. Release the left mouse button.

The selected region of text is highlighted.

Once a region of text is selected, it becomes the primary X selection (see Section 10.3 [Using X Selections], page 73) as well as the Emacs selected region. You can paste it into other X applications and use the options from the **Edit** pull-down menu on it. Since it is also the Emacs region, you can use Emacs region commands on it.

9.3 Additional Mouse Operations

XEmacs also provides the following mouse functions. Most of these are not bound to mouse gestures by default, but they are provided for your customization pleasure. For example, if you wanted ***`shift-left`*** (that is, holding down the ⟨Shift⟩ key and clicking the left mouse button) to delete the character at which you are pointing, then you could do this:

```
(global-set-key '(shift button1) 'mouse-del-char)
```

`mouse-del-char`
Delete the character pointed to by the mouse.

mouse-delete-window
Delete the Emacs window that the mouse is on.

mouse-keep-one-window
Select the Emacs window that the mouse is on, then delete all other windows on this frame.

mouse-kill-line
Kill the line pointed to by the mouse.

mouse-line-length
Print the length of the line indicated by the pointer.

mouse-scroll
Scroll point to the mouse position.

mouse-select
Select the Emacs window the mouse is on.

mouse-select-and-split
Select the Emacs window mouse is on, then split it vertically in half.

mouse-set-mark
Select the Emacs window the mouse is on and set the mark at the mouse position. Display the cursor at that position for a second.

mouse-set-point
Select the Emacs window that the mouse is on and move point to the mouse position.

mouse-track
Make a selection with the mouse. This is the default binding of the left mouse button (⟨button1⟩).

mouse-track-adjust
Extend the existing selection. This is the default binding of ⟨Shift-button1⟩.

mouse-track-and-copy-to-cutbuffer
Make a selection like `mouse-track`, but also copy it to the cut buffer.

mouse-track-delete-and-insert
Make a selection with the mouse and insert it at point. This is the default binding of ⟨control-shift-button1⟩.

mouse-track-insert
Make a selection with the mouse and insert it at point. This is the default binding of ⟨control-button1⟩.

mouse-window-to-region
Narrow a window to the region between the cursor and the mouse pointer.

The *M-x mouse-track* command should be bound to a mouse button. If you click-and-drag, the selection is set to the region between the point of the initial click and the point at which you release the button. These positions do not need to be ordered.

If you click-and-release without moving the mouse, the point is moved, and the selection is disowned (there will be no selection owner.) The mark will be set to the previous position of point.

If you double-click, the selection will extend by symbols instead of by characters. If you triple-click, the selection will extend by lines.

If you drag the mouse off the top or bottom of the window, you can select pieces of text that are larger than the visible part of the buffer; the buffer will scroll as necessary.

The selected text becomes the current X selection, and is also copied to the top of the kill ring. Point will be left at the position at which you released the button and the mark will be left at the initial click position. Bind a mouse click to *mouse-track-and-copy-to-cutbuffer* to copy selections to the cut buffer. (See also the `mouse-track-adjust` command, on *Shift-button1*.)

The *M-x mouse-track-adjust* command should be bound to a mouse button. The selection will be enlarged or shrunk so that the point of the mouse click is one of its endpoints. This is only meaningful after the `mouse-track` command (⟨button1⟩) has been executed.

The *M-x mouse-track-delete-and-insert* command is exactly the same as the `mouse-track` command on ⟨button1⟩, except that point is not moved; the selected text is immediately inserted after being selected; and the text of the selection is deleted.

The *M-x mouse-track-insert* command is exactly the same as the `mouse-track` command on ⟨button1⟩, except that point is not moved; the selected text is immediately inserted after being selected; and the selection is immediately disowned afterwards.

10 Killing and Moving Text

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

The most common way of moving or copying text with Emacs is to kill it and later yank it in one or more places. This is safe because all the text killed recently is stored in the kill ring, and it is versatile, because you can use the same commands for killing syntactic units and for moving those units. There are other ways of copying text for special purposes.

Emacs has only one kill ring, so you can kill text in one buffer and yank it in another buffer. If you are using XEmacs under X, you can also use the X selection mechanism to copy text from one buffer to another, or between applications. See Section 10.3 [Using X Selections], page 73.

10.1 Deletion and Killing

Most commands that erase text from the buffer save it. You can get the text back if you change your mind, or you can move or copy it to other parts of the buffer. Commands which erase text and save it in the kill ring are known as *kill* commands. Some other commands erase text but do not save it; they are known as *delete* commands. (This distinction is made only for erasing text in the buffer.)

The commands' names and individual descriptions use the words '`kill`' and '`delete`' to indicate what they do. If you perform a kill or delete command by mistake, use the *C-x u* (`undo`) command to undo it (see Chapter 5 [Undo], page 49). 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 usually kill.

10.1.1 Deletion

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

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

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

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

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

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

The most basic delete commands are *C-d* (`delete-char`) and ⟨DEL⟩ (`delete-backward-char`). *C-d* deletes the character after point, the one the cursor is "on top of". Point doesn't move. ⟨DEL⟩ deletes the character before the cursor, and moves point back. 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; if you give them an argument, they kill instead, since they can erase more than one character this way.

The other delete commands delete only formatting characters: spaces, tabs and newlines. *M-* (`delete-horizontal-space`) deletes all spaces and tab characters before and after point. *M-⟨SPC⟩* (`just-one-space`) does the same 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 as leaving one blank line, the current line. *M-^* (`delete-indentation`) joins the current line and the previous line, or, if given an argument, joins the current line and the next line by deleting a newline and all surrounding spaces, possibly leaving a single space. See Chapter 20 [Indentation], page 145.

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 the line blank. If given on a blank line, the blank line disappears. As a consequence, a line disappears completely if you go to the front of a non-blank line 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 the line, thus merging the next line into the current one. Emacs ignores invisible spaces and tabs at the end of the line when deciding which case applies: if point appears to be at the end of the line, you can be sure the newline will be killed.

If you give *C-k* a positive argument, it kills that many lines and the newlines that follow them (however, text on the current line before point is not killed). With a negative argument, *C-k* kills back to a number of line beginnings. An argument of −2 means kill back to the second line beginning. If point is at the beginning of a line, that line beginning doesn't count, so *C-u - 2 C-k* with point at the front of a line kills the two previous lines.

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

10.1.3 Other Kill Commands

C-w Kill region (from point to the mark) (`kill-region`). See Section 21.2 [Words], page 156.

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

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

C-x ⟨DEL⟩ Kill back to beginning of sentence (`backward-kill-sentence`). See Section 21.3 [Sentences], page 157.

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

C-w (`kill-region`) is a very general kill command; it kills everything between point and the mark. You can use this command to kill any contiguous sequence of characters by first setting the mark at one end of a sequence of characters, then going to the other end and typing *C-w*.

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

Other syntactic units can be killed: words, with *M-*⟨DEL⟩ and *M-d* (see Section 21.2 [Words], page 156); sexps, with *C-M-k* (see Section 22.2 [Lists], page 164); and sentences, with *C-x* ⟨DEL⟩ and *M-k* (see Section 21.3 [Sentences], page 157).

10.2 Yanking

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

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

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

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

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

10.2.1 The Kill Ring

All killed text is recorded in the *kill ring*, a list of blocks of text that have been killed. There is only one kill ring, used in all buffers, so you can kill text in one buffer and yank it in another buffer. This is the usual way to move text from one file to another. (See Section 10.4 [Accumulating Text], page 76, for some other ways.)

If you have two separate Emacs processes, you cannot use the kill ring to move text. If you are using XEmacs under X, however, you can use the X selection mechanism to move text from one to another.

If you are using XEmacs under X and have one Emacs process with multiple frames, they do share the same kill ring. You can kill or copy text in one Emacs frame, then yank it in the other frame belonging to the same process.

The command *C-y* (`yank`) reinserts the text of the most recent kill. It leaves the cursor at the end of the text and sets the mark at the beginning of the text. See Chapter 9 [Mark], page 63.

C-u C-y yanks the text, leaves the cursor in front of the text, and sets the mark after it, if the argument is with just a *C-u*. Any other argument, including *C-u* and digits, has different results, described below, under "Yanking Earlier Kills".

To copy a block of text, you can also use *M-w* (`copy-region-as-kill`), which copies the region into the kill ring without removing it from the buffer. *M-w* is similar to *C-w* followed by *C-y* but does not mark the buffer as "modified" and does not actually cut anything.

10.2.2 Appending Kills

Normally, each kill command pushes a new block onto the kill ring. However, two or more kill commands in a row combine their text into a single entry, so that a single *C-y* yanks it all back. This means you don't have to kill all the text you want to yank in one command; you can kill line after line, or word after word, until you have killed what you want, then get it all back at once using *C-y*. (Thus we join television in leading people to kill thoughtlessly.)

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

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

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

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

10.2.3 Yanking Earlier Kills

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

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

Yanking moves the "last yank" pointer around the ring, but 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.

Use *M-y* with a numeric argument to advance the "last yank" pointer by the specified number of items. A negative argument moves the pointer toward the front of the ring; from the front of the ring, it moves to the last entry and starts moving forward from there.

Once the text you are looking for is brought into the buffer, you can stop doing *M-y* commands and the text will stay there. Since the text is just a copy of the kill ring item, editing it in the buffer does not change what's in the ring. As long you don't kill additional text, the "last yank" pointer remains at the same place in the kill ring: repeating *C-y* will yank another copy of the same old kill.

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

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

10.3 Using X Selections

In the X window system, mouse selections provide a simple mechanism for text transfer between different applications. In a typical X application, you can select text by pressing the left mouse button and dragging the cursor over the text you want to copy. The text becomes the primary X selection and is highlighted. The highlighted region is also the Emacs selected region.

- Since the region is the primary X selection, you can go to a different X application and click the middle mouse button: the text that you selected in the previous application is pasted into the current application.
- Since the region is the Emacs selected region, you can use all region commands (*C-w, M-w* etc.) as well as the options of the **Edit** menu to manipulate the selected text.

10.3.1 The Clipboard Selection

There are other kinds of X selections besides the **Primary** selection; one common one is the **Clipboard** selection. Some applications prefer to transfer data using this selection in preference to the **Primary**. One can transfer text from the **Primary** selection to the **Clipboard** selection with the **Copy** command under the **Edit** menu in the menubar.

Usually, the clipboard selection is not visible. However, if you run the '`xclipboard`' application, the text most recently copied to the clipboard (with the **Copy** command) is displayed in a window. Any time new text is thus copied, the '`xclipboard`' application makes a copy of it and displays it in its window. The value of the clipboard can survive the lifetime of the running Emacs process. The `xclipboard` man page provides more details.

Warning: If you use the '`xclipboard`' application, remember that it maintains a list of all things that have been pasted to the clipboard (that is, copied with the **Copy** command). If you don't manually delete elements from this list by clicking on the **Delete** button in the `xclipboard` window, the clipboard will eventually consume a lot of memory.

In summary, some X applications (such as '`xterm`') allow one to paste text in them from XEmacs in the following way:

- Drag out a region of text in Emacs with the left mouse button, making that text be the **Primary** selection.
- Click the middle button in the other application, pasting the **Primary** selection.

With some other applications (notably, the OpenWindows and Motif tools) you must use this method instead:

- Drag out a region of text in Emacs with the left mouse button, making that text be the **Primary** selection.
- Copy the selected text to the **Clipboard** selection by selecting the **Copy** menu item from the **Edit** menu, or by hitting the **Copy** key on your keyboard.
- Paste the text in the other application by selecting **Paste** from its menu, or by hitting the **Paste** key on your keyboard.

10.3.2 Miscellaneous X Selection Commands

M-x x-copy-primary-selection
: Copy the primary selection to both the kill ring and the Clipboard.

M-x x-insert-selection
: Insert the current selection into the buffer at point.

M-x x-delete-primary-selection
: Deletes the text in the primary selection without copying it to the kill ring or the Clipboard.

M-x x-kill-primary-selection
: Deletes the text in the primary selection and copies it to both the kill ring and the Clipboard.

M-x x-mouse-kill
: Kill the text between point and the mouse and copy it to the clipboard and to the cut buffer.

M-x x-own-secondary-selection
: Make a secondary X selection of the given argument.

M-x x-own-selection
: Make a primary X selection of the given argument.

M-x x-set-point-and-insert-selection
: Set point where clicked and insert the primary selection or the cut buffer.

10.3.3 X Cut Buffers

X cut buffers are a different, older way of transferring text between applications. XEmacs supports cut buffers for compatibility with older programs, even though selections are now the preferred way of transferring text.

X has a concept of applications "owning" selections. When you select text by clicking and dragging inside an application, the application tells the X server that it owns the selection. When another application asks the X server for the value of the selection, the X server requests the information from the owner. When you use selections, the selection data is not actually transferred unless someone wants it; the act of making a selection doesn't transfer data. Cut buffers are different: when you "own" a cut buffer, the data is actually transferred to the X server immediately, and survives the lifetime of the application.

Any time a region of text becomes the primary selection in Emacs, Emacs also copies that text to the cut buffer. This makes it possible to copy text from an XEmacs buffer and paste it into an older, non-selection-based application (such as Emacs 18).

Note: Older versions of Emacs could not access the X selections, only the X cut buffers.

10.3.4 Active Regions

By default, both the text you select in an Emacs buffer using the click-and-drag mechanism and text you select by setting point and the mark is highlighted. You can use Emacs region commands as well as the **Cut** and **Copy** commands on the highlighted region you selected with the mouse.

If you prefer, you can make a distinction between text selected with the mouse and text selected with point and the mark by setting the variable `zmacs-regions` to `nil`. In that case:

- The text selected with the mouse becomes both the X selection and the Emacs selected region. You can use menu-bar commands as well as Emacs region commands on it.
- The text selected with point and the mark is not highlighted. You can only use Emacs region commands on it, not the menu-bar items.

Active regions originally come from Zmacs, the Lisp Machine editor. The idea behind them is that commands can only operate on a region when the region is in an "active" state. Put simply, you can only operate on a region that is highlighted.

The variable `zmacs-regions` checks whether LISPM-style active regions should be used. This means that commands that operate on the region (the area between point and the mark) only work while the region is in the active state, which is indicated by highlighting. Most commands causes the region to not be in the active state; for example, *`C-w`* only works immediately after activating the region.

More specifically:

- Commands that operate on the region only work if the region is active.
- Only a very small set of commands causes the region to become active— those commands whose semantics are to mark an area, such as `mark-defun`.
- The region is deactivated after each command that is executed, except that motion commands do not change whether the region is active or not.

`set-mark-command` (*`C-SPC`*) pushes a mark and activates the region. Moving the cursor with normal motion commands (*`C-n`*, *`C-p`*, etc.) will cause the region between point and the recently-pushed mark to be highlighted. It will remain highlighted until some non-motion command is executed.

`exchange-point-and-mark` (*C-x C-x*) activates the region. So if you mark a region and execute a command that operates on it, you can reactivate the same region with *C-x C-x* (or perhaps *C-x C-x C-x C-x*) to operate on it again.

Generally, commands that push marks as a means of navigation, such as `beginning-of-buffer` (*M-<*) and `end-of-buffer` (*M->*), do not activate the region. However, commands that push marks as a means of marking an area of text, such as `mark-defun` (*M-C-h*), `mark-word` (*M-@*), and `mark-whole-buffer` (*C-x h*), do activate the region.

When `zmacs-regions` is `t`, there is no distinction between the primary X selection and the active region selected by point and the mark. To see this, set the mark (⟨C-SPC⟩) and move the cursor with any cursor-motion command: the region between point and mark is highlighted, and you can watch it grow and shrink as you move the cursor.

Any other commands besides cursor-motion commands (such as inserting or deleting text) will cause the region to no longer be active; it will no longer be highlighted, and will no longer be the primary selection. Region can be explicitly deactivated with *C-g*.

Commands that require a region (such as *C-w*) signal an error if the region is not active. Certain commands cause the region to be in its active state. The most common ones are `push-mark` (⟨C-SPC⟩) and `exchange-point-and-mark` (*C-x C-x*).

When `zmacs-regions` is `t`, programs can be non-intrusive on the state of the region by setting the variable `zmacs-region-stays` to a non-`nil` value. If you are writing a new Emacs command that is conceptually a "motion" command and should not interfere with the current highlightedness of the region, then you may set this variable. It is reset to `nil` after each user command is executed.

When `zmacs-regions` is `t`, programs can make the region between point and mark go into the active (highlighted) state by using the function `zmacs-activate-region`. Only a small number of commands should ever do this.

When `zmacs-regions` is `t`, programs can deactivate the region between point and the mark by using `zmacs-deactivate-region`. Note: you should not have to call this function; the command loop calls it when appropriate.

10.4 Accumulating Text

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

If you like, you can accumulate blocks of text from scattered locations either into a buffer or into a file. The relevant commands are described here. You can also use Emacs registers for storing and accumulating text. See Chapter 11 [Registers], page 79.

M-x append-to-buffer
: Append region to contents of specified buffer (`append-to-buffer`).

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

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

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

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

To accumulate text into a buffer, use the command *M-x append-to-buffer*, which inserts a copy of the region into the buffer *buffername*, at the location of point in that buffer. If there is no buffer with the given name, one is created.

If you append text to a buffer that has been used for editing, the copied text goes to the place where point is. 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, this command does not always append to the text already in the buffer; but if this command is the only command used to alter a buffer, it does always append to the existing text because point is always at the end.

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

You can retrieve the accumulated text from that buffer with *M-x insert-buffer*, which takes *buffername* as an argument. It inserts a copy of the text in buffer *buffername* into the selected buffer. You could alternatively select the other buffer for editing, perhaps moving text from it by killing or with `append-to-buffer`. See Chapter 16 [Buffers], page 125, for background information on buffers.

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

10.5 Rectangles

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

To specify the rectangle a command should work on, put the mark at one corner and point at the opposite corner. The specified rectangle is called the *region-rectangle* because it is controlled about the same way the region is controlled. Remember that a given combination of point and mark values can be interpreted either as specifying a region or as specifying a rectangle; it is up to the command that uses them to choose the interpretation.

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

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

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

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

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

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

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

Note that "killing" a rectangle is not killing in the usual sense; the rectangle is not stored in the kill ring, but in a special place that only records the most recently killed rectangle (that is, does not append to a killed rectangle). Different yank commands have to be used and only one rectangle is stored, because yanking a rectangle is quite different from yanking linear text and yank-popping commands are difficult to make sense of.

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

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

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

Rectangles can also be copied into and out of registers. See Section 11.3 [Rectangle Registers], page 80.

11 Registers

Emacs *registers* are places in which you can save text or positions for later use. Text saved in a register can be copied into the buffer once or many times; a position saved in a register is used by moving point to that position. Rectangles can also be copied into and out of registers (see Section 10.5 [Rectangles], page 77).

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

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

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

11.1 Saving Positions in Registers

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

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

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

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

The command *C-x r j r* moves point to the location recorded in register *r*. The register is not affected; it continues to record the same location. You can jump to the same position using the same register as often as you want.

11.2 Saving Text in Registers

When you want to insert a copy of the same piece of text many times, it can be impractical to use the kill ring, since each subsequent kill moves the piece of text further down on the ring. It becomes hard to keep track of the argument needed to retrieve the same text with *C-y*. 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 g* (`insert-register`).

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

C-x r g r
: Insert text contents of 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` deletes the text from the buffer as well.

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

11.3 Saving Rectangles in Registers

A register can contain a rectangle instead of lines of text. The rectangle is represented as a list of strings. See Section 10.5 [Rectangles], page 77, for basic information on rectangles and how to specify rectangles in a buffer.

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

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

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

12 Controlling the Display

Since only part of a large buffer fits in the window, XEmacs 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 frame and redisplay, scrolling the selected window to center point vertically within it (`recenter`).

C-v
pgdn
next Scroll forward (a windowful or a specified number of lines) (`scroll-up`). On most X keyboards, you can get this functionality using the key labelled '`Page Down`', which generates either *next* or *pgdn*.

M-v
pgup
prior Scroll backward (`scroll-down`). On most X keyboards, you can get this functionality using the key labelled '`Page Up`', which generates either *prior* or *pgup*.

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

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

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

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

12.1 Scrolling

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

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

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

The most basic scrolling command is *C-l* (`recenter`) with no argument. It clears the entire frame 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. *Meta-v* (`scroll-down`) is like *C-v*, but moves in the opposite direction.

To read the buffer a windowful at a time, use *C-v* with no argument. *C-v* 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. Point moves to the new top of the window if it was in the text scrolled off the top. *M-v* with no argument moves backward with similar overlap. 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 scroll is using *C-l* with a numeric argument. *C-l* does not clear the frame when given an argument; it only scrolls the selected window. With a positive argument *n*, *C-l* 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 frame. *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 frame.

Scrolling happens automatically if point has moved out of the visible portion of the text when it is time to display. Usually scrolling is done to put point vertically centered within the window. However, if the variable `scroll-step` has a non-zero 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 happens.

Scrolling happens automatically if point has moved out of the visible portion of the text when it is time to display. Usually scrolling is done to put point vertically centered within the window. However, if the variable `scroll-step` has a non-zero 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 happens.

If you set `scroll-step` to a small value because you want to use arrow keys to scroll the screen without recentering, the redisplay preemption will likely make XEmacs keep recentering the screen when scrolling fast, regardless of `scroll-step`. To prevent this, set `scroll-conservatively` to a small value, which will have the result of overriding the redisplay preemption.

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 45), with a '`$`' appearing in the first column when there is text truncated to the left, and in the last column when there is text truncated to the right.

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

12.3 Selective Display

XEmacs can hide lines indented more than a certain number of columns (you specify how many columns). This allows you to get an overview of a part of a program.

To hide lines, type *C-x $* (`set-selective-display`) with a numeric argument *n*. (See Section 4.9 [Arguments], page 46, for information on giving the argument.) Lines with at least *n* columns of indentation disappear from the screen. The only indication of their presence are three dots ('`...`'), which appear at the end of each visible line that is followed by one or more invisible ones.

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

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

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

12.4 Variables Controlling Display

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

When you reenter XEmacs after suspending, XEmacs 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 XEmacs is entered and exited, respectively) switch between pages of memory so as to use one page for XEmacs and another page for other output. In that case, you might want to set the variable `no-redraw-on-reenter` to non-`nil` so that XEmacs will assume, when resumed, that the screen page it is using still contains what XEmacs last wrote there.

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

If the variable `ctl-arrow` is `nil`, control characters in the buffer are displayed with octal escape sequences, all except newline and tab. If its value is `t`, then control characters will be printed with an up-arrow, for example *^A*.

If its value is not `t` and not `nil`, then characters whose code is greater than 160 (that is, the space character (32) with its high bit set) will be assumed to be printable, and will be displayed without alteration. This is the default when running under X Windows, since XEmacs assumes an ISO/8859-1 character set (also known as "Latin1"). The `ctl-arrow` variable may also be set to an integer, in which case all characters whose codes are greater than or equal to that value will be assumed to be printable.

Altering the value of `ctl-arrow` makes it local to the current buffer; until that time, the default value is in effect. See Section 29.2.4 [Locals], page 264.

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

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

13 Searching and Replacement

Like other editors, Emacs has commands for searching for occurrences of a string. The principal search command is unusual in that it is *incremental*: it begins to search before you have finished typing the search string. There are also non-incremental 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) is found. When you have typed enough characters to identify the place you want, you can stop. Depending on what you do next, you may or may not need to terminate the search explicitly with a ⟨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.

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

If you make a mistake in typing the search string, you can erase characters with ⟨DEL⟩. Each ⟨DEL⟩ cancels the last character of the 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⟩ (or ⟨C-m⟩), which stops searching, leaving the cursor where the search brought it. Any command not specially meaningful in searches also stops the search and is then executed. Thus, typing *C-a* exits the search and then moves to the beginning of the line. ⟨RET⟩ is necessary only if the next command you want to type is a printing character, ⟨DEL⟩, ⟨ESC⟩, or another control character that is special within searches (*C-q*, *C-w*, *C-r*, *C-s*, or *C-y*).

Sometimes you search for '`FOO`' and find it, but were actually looking for a different occurance of it. To move to the next occurrence of the search string, type another *C-s*. Do this as often as necessary. If you overshoot, you can cancel some *C-s* characters with ⟨DEL⟩.

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

If the specified string is not found at all, the echo area displays the text '`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`', the cursor may be after the '`FOO`' in '`FOOL`'. At this point there are several things you can do. If you mistyped the search string, 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.

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 backward 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. The search is now successful and waiting for more input, so a second `C-g` cancels the entire search.

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

To search backwards, you can use `C-r` instead of `C-s` to start the search; `C-r` is the key that runs the command (`isearch-backward`) to search backward. You can also use `C-r` to change from searching forward to searching backwards. Do this if a search fails because the place you started was too far down in the file. Repeated `C-r` keeps looking for more occurrences backwards. `C-s` starts going forward again. You can cancel `C-r` in a search with ⟨DEL⟩.

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 the rest of the current line into the search string.

The characters `M-p` and `M-n` can be used in an incremental search to recall things which you have searched for in the past. A list of the last 16 things you have searched for is retained, and `M-p` and `M-n` let you cycle through that ring.

The character `M-`⟨TAB⟩ does completion on the elements in the search history ring. For example, if you know that you have recently searched for the string `POTATOE`, you could type `C-s P O M-`⟨TAB⟩. If you had searched for other strings beginning with `PO` then you would be shown a list of them, and would need to type more to select one.

You can change any of the special characters in incremental search via the normal keybinding mechanism: simply add a binding to the `isearch-mode-map`. For example, to make the character *C-b* mean "search backwards" while in isearch-mode, do this:

```
(define-key isearch-mode-map "\C-b" 'isearch-repeat-backward)
```

These are the default bindings of isearch-mode:

DEL Delete a character from the incremental search string (`isearch-delete-char`).

RET Exit incremental search (`isearch-exit`).

C-q Quote special characters for incremental search (`isearch-quote-char`).

C-s Repeat incremental search forward (`isearch-repeat-forward`).

C-r Repeat incremental search backward (`isearch-repeat-backward`).

C-y Pull rest of line from buffer into search string (`isearch-yank-line`).

C-w Pull next word from buffer into search string (`isearch-yank-word`).

C-g Cancels input back to what has been found successfully, or aborts the isearch (`isearch-abort`).

M-p Recall the previous element in the isearch history ring (`isearch-ring-retreat`).

M-n Recall the next element in the isearch history ring (`isearch-ring-advance`).

M-⟨TAB⟩ Do completion on the elements in the isearch history ring (`isearch-complete`).

Any other character which is normally inserted into a buffer when typed is automatically added to the search string in isearch-mode.

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 the search has found. The single-line window appears as soon as point gets outside of the text that is already on the screen.

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

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

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

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

13.2 Non-Incremental Search

Emacs also has conventional non-incremental search commands, which require you 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 non-incremental search, first type *C-s* ⟨RET⟩ (or *C-s C-m*). This enters the minibuffer to read the search string. Terminate the string with ⟨RET⟩ to start the search. If the string is not found, the search command gets an error.

By default, *C-s* invokes incremental search, but if you give it an empty argument, which would otherwise be useless, it invokes non-incremental search. Therefore, *C-s* ⟨RET⟩ invokes non-incremental search. *C-r* ⟨RET⟩ also works this way.

Forward and backward non-incremental searches are implemented by the commands `search-forward` and `search-backward`. You can bind these commands to keys. The reason that incremental search is programmed to invoke them as well is that *C-s* ⟨RET⟩ is the traditional sequence of characters used in Emacs to invoke non-incremental search.

Non-incremental searches performed using *C-s* ⟨RET⟩ do not call `search-forward` right away. They first check if the next character is *C-w*, which requests a word search.

13.3 Word Search

Word search looks 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 is found even if there are multiple spaces, newlines or other punctuation between the words.

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

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

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

Word search is a special case of non-incremental search. It is invoked with *C-s* ⟨RET⟩ *C-w* followed by the search string, which must always be terminated with another ⟨RET⟩. Being non-incremental, 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 89.

You can do a backward word search with *C-r* ⟨RET⟩ *C-w*.

Forward and backward word searches are implemented by the commands `word-search-forward` and `word-search-backward`. You can bind these commands to keys. The reason that incremental search is programmed to invoke them as well is that *C-s* ⟨RET⟩ *C-w* is the traditional Emacs sequence of keys for word search.

13.4 Regular Expression Search

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

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

All of the control characters that do special things within an ordinary incremental search have the same functionality in incremental regexp search. Typing *C-s* or *C-r* immediately after starting a search retrieves the last incremental search regexp used: incremental regexp and non-regexp searches have independent defaults.

Non-incremental search for a regexp is done by the functions `re-search-forward` and `re-search-backward`. You can invoke them with *M-x* or bind them to keys. You can also call `re-search-forward` by way of incremental regexp search with *M-C-s* ⟨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 character and nothing else. The special characters are '`$`', '`^`', '`.`', '`*`', '`+`', '`?`', '`[`', '`]`' and '`\`'; no new special characters will be defined. Any other character appearing in a regular expression is ordinary, unless a '`\`' precedes it.

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

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

As a simple example, you can concatenate the regular expressions '`f`' and '`o`' to get the regular expression '`fo`', which matches only the string '`fo`'. To do something nontrivial, you need to use one of the following special characters:

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

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

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

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

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

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

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

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

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

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

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

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

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

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

\ does two things: it quotes the special characters (including '\'), and it introduces additional special constructs.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

`\s`*code* matches any character whose syntax is *code*. *code* is a character which represents a syntax code: thus, '`w`' for word constituent, '`-`' for whitespace, '`(`' for open-parenthesis, etc. See Section 29.5 [Syntax], page 274.

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

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

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

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

13.6 Searching and Case

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

If you want a case-sensitive search, set the variable `case-fold-search` to `nil`. Then all letters must match exactly, including case. `case-fold-search` is a per-buffer variable; altering it affects only the current buffer, but there is a default value which you can change as well. See Section 29.2.4 [Locals], page 264. You can also use **Case Sensitive Search** from the **Options** menu on your screen.

13.7 Replacement Commands

Global search-and-replace operations are not needed as often in Emacs as they are in other editors, but they are available. In addition to the simple `replace-string` command which is like that found in most editors, there is a `query-replace` command which asks you, for each occurrence of a 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 204.

13.7.1 Unconditional Replacement

M-x replace-string ⟨RET⟩ *string* ⟨RET⟩ *newstring* ⟨RET⟩
    Replace every occurrence of *string* with *newstring*.

M-x replace-regexp ⟨RET⟩ *regexp* ⟨RET⟩ *newstring* ⟨RET⟩
    Replace every match for *regexp* with *newstring*.

To replace every instance of '`foo`' after point with '`bar`', use the command *M-x replace-string* with the two arguments '`foo`' and '`bar`'. Replacement occurs only after point: if you want to cover the whole buffer you must go to the beginning first. By default, 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 28.3 [Narrowing], page 252).

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

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

13.7.2 Regexp Replacement

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

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

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

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

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

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

13.7.3 Replace Commands and Case

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

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

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

13.7.4 Query Replace

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

M-x query-replace-regexp ⟨RET⟩ *regexp* ⟨RET⟩ *newstring* ⟨RET⟩
Replace some matches for *regexp* with *newstring*.

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

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

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

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

⟨DEL⟩
to skip to the next occurrence without replacing this one.

, (Comma)
to replace this occurrence and display the result. You are then prompted for another input character. However, since the replacement has already been made, ⟨DEL⟩ and ⟨SPC⟩ are equivalent. At this point, you can type *C-r* (see below) to alter the replaced text. To undo the replacement, you can type *C-x u*. This exits the `query-replace`. If you want to do further replacement you must use *C-x ESC* to restart (see Section 6.4 [Repetition], page 55).

⟨ESC⟩
to exit without doing any more replacements.

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

!
to replace all remaining occurrences without asking again.

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

C-r
to enter a recursive editing level, in case the occurrence needs to be edited rather than just replaced with *newstring*. When you are done, exit the recursive editing level with *C-M-c* and the next occurrence will be displayed. See Section 28.5 [Recursive Edit], page 253.

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

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

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

If you type any other character, Emacs exits the `query-replace`, and executes the character as a command. To restart the `query-replace`, use `C-x` ⟨ESC⟩, which repeats the `query-replace` because it used the minibuffer to read its arguments. See Section 6.4 [Repetition], page 55.

13.8 Other Search-and-Loop Commands

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

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

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

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

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

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

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

14 Commands for Fixing Typos

This chapter describes commands that are especially useful when you catch a mistake in your text just after you have made it, or when you change your mind while composing text on line.

14.1 Killing Your Mistakes

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

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

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

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

When your mistake is longer than a couple of characters, it might be more convenient to use *M-*⟨DEL⟩ or *C-x* ⟨DEL⟩. *M-*⟨DEL⟩ kills back to the start of the last word, and *C-x* ⟨DEL⟩ kills back to the start of the last sentence. *C-x* ⟨DEL⟩ is particularly useful when you are thinking of what to write as you type it, in case you change your mind about phrasing. *M-*⟨DEL⟩ and *C-x* ⟨DEL⟩ save the killed text for *C-y* and *M-y* to retrieve. See Section 10.2 [Yanking], page 71.

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.

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 adjacent characters can be fixed 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, *C-t* transposes the last two characters on the line, rather than transposing the last character of the line with the newline, which would be useless. If you catch a transposition error right away, you can fix it with just *C-t*. If you catch the error later, 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 85.

Meta-t (`transpose-words`) transposes the word before point with the word after point. It moves point forward over a word, dragging the word preceding or containing point forward as well.

The punctuation characters between the words do not move. For example, 'FOO, BAR' transposes into 'BAR, FOO' rather than 'BAR FOO,'.

C-M-t (transpose-sexps) is a similar command for transposing two expressions (see Section 22.2 [Lists], page 164), and *C-x C-t* (transpose-lines) exchanges lines. It works like *M-t* but in determines the division of the text into syntactic units differently.

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

A numeric argument of zero transposes the character (word, sexp, line) ending after point with the one ending after the mark (otherwise a command with a repeat count of zero would do nothing).

14.3 Case Conversion

M-- M-l Convert last word to lower case. Note that *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 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* do not move the cursor when used with a negative argument. As soon as you see you have mistyped the last word, you can simply case-convert it and continue typing. See Section 21.7 [Case], page 162.

14.4 Checking and Correcting Spelling

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

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

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

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

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

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

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

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

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

15 File Handling

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

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

15.1 File Names

Most Emacs commands that operate on a file require you to specify the file name. (Saving and reverting are exceptions; the buffer knows which file name to use for them.) File names are specified in the minibuffer (see Chapter 6 [Minibuffer], page 51). *Completion* is available, to make it easier to specify long file names. See Section 6.3 [Completion], page 53.

There is always a *default file name* which is used if you enter an empty argument by typing just ⟨RET⟩. Normally the default file name is the name of the file visited in the current buffer; this makes it easy to operate on that file with any of the Emacs file commands.

Each buffer has a default directory, normally the same as the directory of the file visited in that buffer. When Emacs reads a file name, the default directory is used if you do not specify a directory. 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 of the current buffer is kept in the variable `default-directory`, which has a separate value in every buffer. The value of the variable should end with a slash.

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

The variable `default-directory-alist` takes an alist of major modes and their opinions on `default-directory` as a Lisp expression to evaluate. A resulting value of `nil` is ignored in favor of `default-directory`.

You can create a new directory with the function `make-directory`, which takes as an argument a file name string. The current directory is displayed in the minibuffer when the function is called; you can delete the old directory name and supply a new directory name. For example, if the current directory is '`/u/rms/gnu`', you can delete '`gnu`' and type '`oryx`' and ⟨RET⟩ to create '`/u/rms/oryx`'. Removing a directory is similar to creating one. To remove a directory, use `remove-directory`; it takes one argument, a file name string.

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

The default directory name 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. To inhibit the insertion of the default directory, set the variable `insert-default-directory` 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. The final minibuffer contents may look invalid, but that is not so. See Section 6.1 [Minibuffer File], page 51.

'`$`' in a file name is used to substitute environment variables. For example, if you have used the shell command '`setenv FOO rms/hacks`' to set up an environment variable named '`FOO`', then you can use '`/u/$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 '`$`' at the same time variable substitution is performed for single '`$`'. The Lisp function that performs the substitution is called `substitute-in-file-name`. The substitution is performed only on filenames read as such using the minibuffer.

15.2 Visiting Files

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

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

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

C-x 5 C-f Visit a file, in another frame (`find-file-other-frame`). Don't change this window or frame.

Visiting a file means copying its contents into an Emacs buffer so you can edit it. Emacs creates a new buffer for each file you visit. We say that the buffer is visiting the file that it was created to hold. Emacs constructs the buffer name from the file name by throwing away the directory and keeping just the file name. For example, a file named '`/usr/rms/emacs.tex`' is displayed in a buffer named '`emacs.tex`'. If a buffer with that name exists, a unique name is constructed by appending '`<2>`', '`<3>`',and 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 visit, or any other permanent place, 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 104.

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

To visit a file, use the command *C-x C-f* (`find-file`). Follow the command with the name of the file you wish to visit, terminated by a ⟨RET⟩. If you are using XEmacs under X, you can also use the **Open...** command from the **File** menu bar item.

The file name is read using the minibuffer (see Chapter 6 [Minibuffer], page 51), with defaulting and completion in the standard manner (see Section 15.1 [File Names], page 101). While in the minibuffer, you can abort *C-x C-f* by typing *C-g*.

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

If you visit a file that is already in Emacs, *C-x C-f* does not make another copy. It selects the existing buffer containing that file. However, before doing so, it checks that the file itself has not changed since you visited or saved it last. If the file has changed, Emacs prints a warning message. See Section 15.3.2 [Simultaneous Editing], page 107.

You can switch to a specific file called out in the current buffer by calling the function `find-this-file`. By providing a prefix argument, this function calls `filename-at-point` and switches to a buffer visiting the file *filename*. It creates one if none already exists. You can use this function to edit the file mentioned in the buffer you are working in or to test if the file exists. You can do that by using the minibuffer completion after snatching the all or part of the filename.

If the variable `find-file-use-truenames`'s value is non-`nil`, a buffer's visited filename will always be traced back to the real file. The filename will never be a symbolic link, and there will never be a symbolic link anywhere in its directory path. In other words, the `buffer-file-name` and `buffer-file-truename` will be equal.

If the variable `find-file-compare-truenames` value is non-`nil`, the `find-file` command will check the `buffer-file-truename` of all visited files when deciding whether a given file is already in a buffer, instead of just `buffer-file-name`. If you attempt to visit another file which is a hard-link or symbolic-link to a file that is already in a buffer, the existing buffer will be found instead of a newly created one.

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

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

If the file you specify is actually a directory, Dired is called on that directory (see Section 15.9 [Dired], page 120). To inhibit this, set the variable `find-file-run-dired` to `nil`; then it is an error to try to visit a directory.

C-x 4 f (`find-file-other-window`) is like *C-x C-f* except that the buffer containing the specified file is selected in another window. The window that was selected before *C-x 4 f* continues to show the same buffer it was already showing. If you use this command 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 131.

C-x 5 C-f (`find-file-other-frame`) is like *C-x C-f* except that it creates a new frame in which the file is displayed.

Use the function `find-this-file-other-window` to edit a file mentioned in the buffer you are editing or to test if that file exists. To do this, use the minibuffer completion after snatching the part or all of the filename. By providing a prefix argument, the function calls `filename-at-point` and switches you to a buffer visiting the file *filename* in another window. The function creates a buffer if none already exists. This function is similar to `find-file-other-window`.

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

15.3 Saving Files

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

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

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

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

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

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

To save a 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), Emacs does not save it because it would have no effect. Instead, *C-x C-s* prints a message in the echo area saying:

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

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

If you have changed a buffer and do not want the changes to be saved, you should take some action to prevent it. Otherwise, you are liable to save it by mistake each time you use `save-some-buffers` or a related command. One thing you can do is type *M-~* (`not-modified`), which removes the indication that the buffer is modified. If you do this, none of the save commands will believe that the buffer needs to be saved. ('~' is often used as a mathematical symbol for 'not'; thus *Meta-~* is 'not', metafied.) You could also use `set-visited-file-name` (see below) to mark the buffer as visiting a different file name, not in use for anything important.

You can also undo all the changes made since the file was visited or saved, by reading the text from the file again. This is called *reverting*. See Section 15.4 [Reverting], page 108. Alternatively, you can undo all the changes by repeating the undo command *C-x* `u`; but this only works if you have not made more changes than the undo mechanism can remember.

M-x set-visited-file-name alters the name of the file that the current buffer is visiting. It prompts you for the new file name in the minibuffer. You can also use `set-visited-file-name` on a buffer that is not visiting a file. The buffer's name is changed to correspond to the file it is now visiting unless the new name is already used by a different buffer; in that case, the buffer name is not changed. `set-visited-file-name` does not save the buffer in the newly visited file; it just alters the records inside Emacs so that it will save the buffer in that file. It also marks the buffer as "modified" so that *C-x C-s will* save.

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

If 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 107.

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.

Use the hook variable `write-file-hooks` to implement other ways to write files, and specify things to be done before files are written. 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. Use the hook variable `after-save-hook` to list all the functions to be called after writing out a buffer to a file.

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. (Make sure 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 you edit.

Emacs makes a backup for a file only the first time a 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 is made by the next save.

15.3.1.1 Single or Numbered Backups

If you choose to have a single backup file (the default), the backup file's name is constructed by appending '`~`' to the file name being edited; thus, the backup file for '`eval.c`' is '`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 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`
: Never make numbered backups; always make single backups.

`version-control` may be set locally in an individual buffer to control the making of backups for that buffer's file. For example, Rmail mode locally sets `version-control` to `never` to make sure that there is only one backup for an Rmail file. See Section 29.2.4 [Locals], page 264.

15.3.1.2 Automatic Deletion of Backups

To prevent unlimited consumption of disk space, Emacs can delete numbered backup versions automatically. Generally Emacs keeps the first few backups and the latest few backups, deleting any in between. This happens every time a new backup is made. The two variables that control the deletion are `kept-old-versions` and `kept-new-versions`. Their values are, respectively the number of oldest (lowest-numbered) backups to keep and the number of newest (highest-numbered) ones to keep, each time a new backup is made. The 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`, excess middle versions are deleted without notification. If it is `nil`, the default, you are asked whether the excess middle versions should really be deleted.

You can also use Dired's `.` (Period) command to delete old versions. See Section 15.9 [Dired], page 120.

15.3.1.3 Copying vs. Renaming

You can make backup files by copying the old file or by renaming it. This makes a difference when the old file has multiple names. If you rename the old file into the backup file, the alternate names become names for the backup file. If you copy the old file instead, the alternate names remain names for the file that you are editing, and the contents accessed by those names will be the new contents.

How you make a backup file may also affect the file's owner and group. If you use copying, they 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 is always the person who last edited the file. Occasionally there is a file whose owner should not change. Since most files should change owners, it is a good idea to use local variable lists to set `backup-by-copying-when-mismatch` for the special cases where the owner should not change (see Section 29.2.5 [File Variables], page 266).

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

15.3.2 Protection Against Simultaneous Editing

Simultaneous editing occurs when two users visit the same file, both make changes, and both save their changes. If no one was informed that this was happening, and you saved first, you would later find that your changes were lost. On some systems, Emacs notices immediately when the second user starts to change a file already being edited, and issues a warning. When this is not possible, or if the second user has started to change the file despite the warning, Emacs checks when the file is saved, and issues a second warning when a user is about to overwrite a file containing another user's changes. If you are the user editing the file, you can take corrective action at this point and prevent actual loss of work.

When you make the first modification in an Emacs buffer that is visiting a file, Emacs records that you have locked the file. (It does this by writing another file in a directory reserved for this purpose.) The lock is removed when you save the changes. The idea is that the file is locked whenever the buffer is modified. If you begin to modify the buffer while the visited file is locked by someone else, this constitutes a collision, and Emacs asks you what to do. It does this by calling the Lisp function `ask-user-about-lock`, which you can redefine to customize what it does. 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 get 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 do 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 can still detect it in time to prevent you from overwriting someone else's changes.

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

The first thing you should do when notified that simultaneous editing has already taken place is to list the directory with *C-u C-x C-d* (see Section 15.7 [Directory Listing], page 119). This will show the file's current author. You should attempt to contact that person and ask 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 a file that is already visited with *C-x C-f* and when you start to modify a file. This is not strictly necessary, but it is useful to find out about such a problem as early as possible, when corrective action takes less work.

Another way to protect your file is to set the read, write, and executable permissions for the file. Use the function `set-default-file-modes` to set the UNIX `umask` value to the *nmask* argument. The `umask` value is the default protection mode for new files.

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 all changes by reading in the previous version of the file. To do this, use *M-x revert-buffer*, which operates on the current buffer. Since reverting a buffer can result in very extensive changes, you must confirm it with *yes*.

If the current buffer has been auto-saved more recently than it has been saved explicitly, `revert-buffer` offers to read the auto save file instead of the visited file (see Section 15.5 [Auto Save], page 109). Emacs asks you about the auto-save file before the request for confirmation of the *revert-buffer* operation, and demands *y* or *n* as an answer. If you have started to type *yes* for confirmation without realizing that the auto-save question was going to be asked, the *y* will answer that question, but the *es* will not be valid confirmation. This gives you a chance to cancel the operation with *C-g* and try again with the answers you really intend.

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

A buffer reverted from its visited file is marked "not modified" until you make a change.

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. 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 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 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 write to the files you visited, because it can be undesirable to save a program that is in an inconsistent state when you have made only 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 save explicitly, for example, with *C-x C-s*.

Normally, the name of the auto-save file is generated by appending '#' to the front and back of the visited file name. Thus, a buffer visiting file 'foo.c' would be auto-saved in a file '#foo.c#'. Most buffers that are not visiting files are auto-saved only if you request it explicitly; when they are auto-saved, the auto-save file name is generated by appending '#%' to the front and '#' to the back of buffer name. For example, the '*mail*' buffer in which you compose messages to be sent is auto-saved in a file named '#%*mail*#'. Names of auto-save files are generated this way unless you customize the functions make-auto-save-file-name and auto-save-file-name-p to do something different. The file name to be used for auto-saving a buffer is calculated at the time 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.

Emacs deletes a buffer's auto-save file when you explicitly save the buffer. To inhibit the deletion, 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 correspond to the new visited name.

15.5.2 Controlling Auto-Saving

Each time you visit a file, auto-saving is turned on for that file's buffer if the variable `auto-save-default` is non-`nil` (but not in batch mode; see Chapter 3 [Entering Emacs], page 33). The default for this variable is `t`, so Emacs auto-saves buffers that visit files by default. You can use the command *M-x auto-save-mode* to turn auto-saving for a buffer on or off. 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 performs auto-saving periodically based on counting how many characters you have typed since the last time auto-saving happened. The variable `auto-save-interval` specifies the number of characters between auto-saves. By default, it is 300. Emacs also auto-saves whenever you call the function `do-auto-save`.

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

You can set the number of seconds of idle time before an auto-save is done. Setting the value of the variable `auto-save-timeout` to zero or `nil` will disable auto-saving due to idleness.

The actual amount of idle time between auto-saves is logarithmically related to the size of the current buffer. This variable is the number of seconds after which an auto-save will happen when the current buffer is 50k or less; the timeout will be 2 1/4 times this in a 200k buffer, 3 3/4 times this in a 1000k buffer, and 4 1/2 times this in a 2000k buffer.

For this variable to have any effect, you must do `(require 'timer)`.

15.5.3 Recovering Data from Auto-Saves

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

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

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

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

15.6 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.

15.6.1 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`).

A *snapshot* is a coherent collection of versions of the various files that make up a program. See Section 15.6.9 [Snapshots], page 117.

15.6.2 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 requires 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.

15.6.3 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.

15.6.4 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`.

15.6.5 Change Logs and VC

If you use RCS for a program and also maintain a change log file for it (see Section 22.10 [Change Log], page 175), 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`'.

15.6.6 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 15.6.9 [Snapshots], page 117) 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 `vcdiff` 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 15.8 [Comparing Files], page 120, for more information about *M-x diff*.

15.6.7 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.

15.6.8 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 15.6.9 [Snapshots], page 117) that mention the file, so that they use the new name; despite this, the snapshot thus modified may not completely work (see Section 15.6.9.2 [Snapshot Caveats], page 117).

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

15.6.9 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.

15.6.9.1 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 15.6.6 [Old Versions], page 115). 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.

15.6.9.2 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.

A snapshot is a set of checked-in versions. So make sure that all the files are checked in and not locked when you make a snapshot.

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.

15.6.10 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. (The actual strings inserted do not have the backslashes in them. They were placed in the Info source file so that the strings don't get interpreted as version-control headers when the Info source files are maintained under version control.) 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 */
```

15.7 Listing a File Directory

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

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

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

To print a directory listing, use *C-x C-d* (`list-directory`). This command prompts in the minibuffer for a file name which is either a directory to be listed or pattern containing wildcards 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`).

Emacs obtains the text of a directory listing 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). `list-directory-verbose-switches` is a string giving the switches to use in a verbose listing (`"-l"` by default).

The variable `directory-abbrev-alist` is an alist of abbreviations for file directories. The list consists of elements of the form `(FROM . TO)`, each meaning to replace `FROM` with `TO` when it appears in a directory name. This replacement is done when setting up the default directory of a newly visited file. Every `FROM` string should start with "^".

Use this feature when you have directories which you normally refer to via absolute symbolic links. Make `TO` the name of the link, and `FROM` the name it is linked to.

15.8 Comparing Files

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

The buffer '`*Diff*`' has Compilation mode as its major mode, so you can use *C-x `* to visit successive changed locations in the two source files. You can also move to a particular hunk of changes and type *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 189.

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 131.

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.

15.9 Dired, the Directory Editor

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

15.9.1 Entering Dired

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

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

15.9.2 Editing in Dired

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

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

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

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

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

15.9.3 Deleting Files With Dired

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

d Flag this file for deletion.

u Remove deletion-flag on this line.

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

x Delete the files that are flagged for deletion.

Flag all auto-save files (files whose names start and end with '`#`') for deletion (see Section 15.5 [Auto Save], page 109).

~ Flag all backup files (files whose names end with '`~`') for deletion (see Section 15.3.1 [Backup], page 105).

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

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

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

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

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

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

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

15.9.4 Immediate File Operations in Dired

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

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

`f` Visits the file described on the current line. It is just like typing `C-x C-f` and supplying that file name. If the file on this line is a subdirectory, `f` actually causes Dired to be invoked on that subdirectory. See Section 15.2 [Visiting], page 102.

`o` Like `f`, but uses another window to display the file's buffer. The Dired buffer remains visible in the first window. This is like using `C-x 4 C-f` to visit the file. See Chapter 17 [Windows], page 131.

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

`v` Views the file described on this line using `M-x view-file`. Viewing a file is like visiting it, but is slanted toward moving around in the file conveniently and does not allow changing the file. See Section 15.10 [Misc File Ops], page 123. Viewing a file that is a directory runs Dired on that directory.

15.10 Miscellaneous File Operations

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

You can use the command *M-x* ***add-name-to-file*** to add a name to an existing file without removing the old name. The new name must belong on the file system that the file is on.

M-x ***append-to-file*** adds the text of the region to the end of the specified file.

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 overwrites the old contents of the file *new*.

M-x ***delete-file*** deletes a specified file, like the `rm` command in the shell. If you are deleting many files in one directory, it may be more convenient to use Dired (see Section 15.9 [Dired], page 120).

M-x ***insert-file*** inserts a copy of the contents of a specified file into the current buffer at point, leaving point unchanged before the contents and the mark after them. See Chapter 9 [Mark], page 63.

M-x ***make-symbolic-link*** reads two file names *old* and *linkname*, and then creates a symbolic link named *linkname* and pointing at *old*. Future attempts to open file *linkname* will then refer to the file named *old* at the time the opening is done, or will result in an error if the name *old* is not in use at that time. Confirmation is required if you create the link while *linkname* is in use. Note that not all systems support symbolic links.

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 previous meaning of the name *new* to be lost. If *old* and *new* are on different file systems, the file *old* is copied and deleted.

M-x ***view-file*** allows you to scan or read a file by sequential screenfuls. It reads a file name argument using the minibuffer. After reading the file into an Emacs buffer, `view-file` reads and displays one windowful. You can then type ⟨SPC⟩ to scroll forward one window, 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 a file for a list of them. Most commands are the default Emacs cursor motion commands. To exit from viewing, type ***C-c***.

16 Using Multiple Buffers

Text you are editing in Emacs resides in an object called a *buffer*. Each time you visit a file, Emacs creates a buffer to hold the file's text. Each time you invoke Dired, Emacs creates a buffer 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, it appears in a buffer called '`*Help*`'.

At any time, one and only one buffer is *selected*. It is also called the *current buffer*. Saying a command operates on "the buffer" really means that the command operates on the selected buffer, as most commands do.

When Emacs creates 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 the window is displaying (see Chapter 17 [Windows], page 131).

Each buffer has a name which can be of any length but is case-sensitive. You can select a buffer using its name. Most buffers are created when you visit files; their names are derived from the files' names. You can also create an empty buffer with any name you want. A newly started Emacs has a buffer named '`*scratch*`' which you can use for evaluating Lisp expressions in Emacs.

Each buffer records 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 143). 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 29.2.4 [Locals], page 264.

16.1 Creating and Selecting Buffers

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

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

`M-x switch-to-other-buffer` *n*
: Switch to the previous buffer.

To select a buffer named *bufname*, type `C-x b` *bufname* ⟨RET⟩. This is the command `switch-to-buffer` with argument *bufname*. You can use completion on an abbreviation for the buffer name you want (see Section 6.3 [Completion], page 53). An empty argument to `C-x b` specifies the most recently selected buffer that is not displayed in any window.

Most buffers are created when you visit files, or use Emacs commands that display text. You can also create a buffer explicitly by typing `C-x b` *bufname* ⟨RET⟩, which creates a new, empty buffer that is not visiting any file, and selects it for editing. The new buffer's major mode is determined by the value of `default-major-mode` (see Chapter 19 [Major Modes], page 143). Buffers not visiting files are usually used for making notes to yourself. If you try to save one, you are asked for the file name to use.

The function `switch-to-buffer-other-frame` is similar to `switch-to-buffer` except that it creates a new frame in which to display the selected buffer.

Use *M-x switch-to-other-buffer* to visit the previous buffer. If you supply a positive integer *n*, the *n*th most recent buffer is displayed. If you supply an argument of 0, the current buffer is moved to the bottom of the buffer stack.

Note that you can also use *C-x C-f* and any other command for visiting a file to switch buffers. See Section 15.2 [Visiting], page 102.

16.2 Listing Existing Buffers

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

To print a list of all existing buffers, type *C-x C-b*. Each line in the list shows one buffer's name, major mode, and visited file. A '`*`' at the beginning of a line indicates the buffer has been "modified". If several buffers are modified, it may be time to save some with *C-x s* (see Section 15.3 [Saving], page 104). A '`%`' indicates a read-only buffer. A '`.`' 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/`'.

As you move the mouse over the '`*Buffer List*`' buffer, the lines are highlighted. This visual cue indicates that clicking the right mouse button (`button3`) will pop up a menu of commands on the buffer represented by this line. This menu duplicates most of those commands which are bound to keys in the '`*Buffer List*`' buffer.

16.3 Miscellaneous Buffer Operations

C-x C-q Toggle read-only status of buffer (`toggle-read-only`).

M-x rename-buffer
: Change the name of the current buffer.

M-x view-buffer
: Scroll through a buffer.

A buffer can be *read-only*, which means that commands to change its text are not allowed. Normally, read-only buffers are created by subsystems such as Dired and Rmail that have special commands to operate on the text. Emacs also creates a read-only buffer if you visit a file that is protected. To make changes in a read-only buffer, use the command *C-x C-q* (`toggle-read-only`). It makes a read-only buffer writable, and makes a writable buffer read-only. This works by setting the variable `buffer-read-only`, which has a local value in each buffer and makes a buffer read-only if its value is non-`nil`.

M-x rename-buffer changes the name of the current buffer, prompting for the new name in the minibuffer. There is no default. If you specify a name that is used by a different buffer, an error is signalled and renaming is not done.

M-x view-buffer is similar to *M-x view-file* (see Section 15.10 [Misc File Ops], page 123), but it examines an already existing Emacs buffer. View mode provides convenient commands for scrolling through the buffer but not for changing it. When you exit View mode, the resulting value of point remains in effect.

To copy text from one buffer to another, use the commands *M-x append-to-buffer* and *M-x insert-buffer*. See Section 10.4 [Accumulating Text], page 76.

16.4 Killing Buffers

After using Emacs for a while, you may accumulate a large number of buffers and may want to eliminate the ones you no longer need. There are several commands for doing this.

C-x k Kill a buffer, specified by name (`kill-buffer`).

M-x kill-some-buffers
: Offer to kill each buffer, one by one.

C-x k (`kill-buffer`) kills one buffer, whose name you specify in the minibuffer. If you type just ⟨RET⟩ in the minibuffer, the default, killing the current buffer, is used. If the current buffer is killed, the buffer that has been selected recently but does not appear in any window now is selected. If the buffer being killed contains unsaved changes, you are asked to confirm with *yes* before the buffer is killed.

The command *M-x kill-some-buffers* asks about each buffer, one by one. An answer of *y* means to kill the buffer. Killing the current buffer or a buffer containing unsaved changes selects a new buffer or asks for confirmation just like `kill-buffer`.

16.5 Operating on Several Buffers

The *buffer-menu* facility is like a "Dired for buffers"; it allows you to request operations on various Emacs buffers by editing a 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. You can only change it using the special commands described in this section. Most of the commands are graphic characters. You can use Emacs cursor motion commands in the '`*Buffer List*`' buffer. If the cursor is on a line describing a buffer, the following special commands apply to that buffer:

d Request to delete (kill) the buffer, then move down. A '`D`' before the buffer name on a line indicates a deletion request. Requested deletions actually take place when you use the *x* command.

k Synonym for *d*.

C-d Like *d* but move up afterwards instead of down.

s Request to save the buffer. An '`S`' befor the buffer name on a line indicates the request. Requested saves actually take place when you use the *x* command. You can request both saving and deletion for the same buffer.

~ Mark buffer "unmodified". The command *~* does this immediately when typed.

x Perform previously requested deletions and saves.

u Remove any request made for the current line, and move down.

⟨DEL⟩ Move to previous line and remove any request made for that line.

All commands that add or remove flags to request later operations also move down a line. They accept a numeric argument as a repeat count, unless otherwise specified.

There are also special commands to use the buffer list to select another buffer, and to specify one or more other buffers for display in additional windows.

1 Select the buffer in a full-frame window. This command takes effect immediately.

2 Immediately set up two windows, with this buffer in one and the buffer selected before '`*Buffer List*`' in the other.

f Immediately select the buffer in place of the '`*Buffer List*`' buffer.

o Immediately select the buffer in another window as if by *C-x 4 b*, leaving '`*Buffer List*`' visible.

q Immediately select this buffer, and display any buffers previously flagged with the *m* command in other windows. If there are no buffers flagged with *m*, this command is equivalent to *1*.

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

Going back between a `buffer-menu` buffer and other Emacs buffers is easy. You can, for example, switch from the '`*Buffer List*`' buffer to another Emacs buffer, and edit there. You can then reselect the `buffer-menu` buffer and perform operations already requested, or you can kill that buffer or pay no further attention to it. All that `buffer-menu` does directly is create and select a suitable buffer, and turn on Buffer Menu mode. All the other capabilities of the buffer menu are implemented by special commands provided in Buffer Menu mode.

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 the commands described here.

17 Multiple Windows

Emacs can split the frame into two or many windows, which can display parts of different buffers or different parts of one buffer. If you are running XEmacs under X, that means you can have the X window that contains the Emacs frame have multiple subwindows.

17.1 Concepts of Emacs Windows

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

At any time, one window is the *selected window*; the buffer displayed by that window is the current buffer. The cursor shows the location of point in that window. Each other window has a location of point as well, but since the terminal has only one cursor, it cannot show the location of point in the other windows.

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, including those 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 non-selected window without affecting the selected window.

Each window has its own mode line, which displays the buffer name, modification status, and major and minor modes of the buffer that is displayed in the window. See Section 1.3 [Mode Line], page 15, for 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`).

`C-x 6` Save the current window configuration in register *reg* (a letter).

`C-x 7` Restore (make current) the window configuration in register *reg* (a letter). Use with a register previously set with `C-x 6`.

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 each of the two windows gets 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 frame have truncated mode lines which do not always appear in inverse video, because Emacs display routines cannot 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, many text lines are too long to fit. Continuing all those lines might be confusing. Set the variable `truncate-partial-width-windows` to non-`nil` to force truncation in all windows less than the full width of the frame, independent of the buffer and its value for `truncate-lines`. See Section 4.7 [Continuation Lines], page 45.

Horizontal scrolling is often used in side-by-side windows. See Chapter 12 [Display], page 81.

You can resize a window and store that configuration in a register by supplying a *register* argument to `window-configuration-to-register` (*C-x 6*). To return to the window configuration established with `window-configuration-to-register`, use `jump-to-register` (*C-x j*).

17.3 Using Other Windows

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

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

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

M-x other-window-any-frame n
: Select the *n*th different window on any frame.

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, the command moves through all the windows in a cyclic order, generally top to bottom and left to right. From the rightmost and bottommost window, it goes back to the one at the upper left corner. A numeric argument, *n*, moves several steps in the cyclic order of windows. A negative numeric argument moves around the cycle in the opposite order. If the optional second argument *all-frames* is non-`nil`, the function cycles through all frames. 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 52.

The command *M-x other-window-any-frame* also selects the window *n* steps away in the cyclic order. However, unlike `other-window`, this command selects a window on the next or previous frame instead of wrapping around to the top or bottom of the current frame, when there are no more windows.

The usual scrolling commands (see Chapter 12 [Display], page 81) apply to the selected window only. *M-C-v* (`scroll-other-window`) scrolls the window that *C-x o* would select. Like *C-v*, it takes positive and negative arguments.

The command *M-x compare-windows* compares the text in the current window with the text in the next window. Comparison starts at point in each window. Point moves forward in each window, a character at a time, until the next set of characters in the two windows are different. Then the command is finished.

A prefix argument *ignore-whitespace* means ignore changes in whitespace. The variable `compare-windows-whitespace` controls how whitespace is skipped.

If `compare-ignore-case` is non-`nil`, changes in case are also ignored.

17.4 Displaying in Another Window

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

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

C-x 4 f filename ⟨RET⟩
: Visit file *filename* and select its buffer in another window. This runs `find-file-other-window`. See Section 15.2 [Visiting], page 102.

C-x 4 d directory ⟨RET⟩
: Select a Dired buffer for directory *directory* in another window. This runs `dired-other-window`. See Section 15.9 [Dired], page 120.

C-x 4 m
: Start composing a mail message in another window. This runs `mail-other-window`, and its same-window version is *C-x m* (see Chapter 26 [Sending Mail], page 211).

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 22.11 [Tags], page 176).

If the variable `display-buffer-function` is non-`nil`, its value is the function to call to handle `display-buffer`. It receives two arguments, the buffer and a flag that if non-`nil` means that the currently selected window is not acceptable. Commands such as `switch-to-buffer-other-window` and `find-file-other-window` work using this function.

17.5 Deleting and Rearranging Windows

C-x 0
: Get rid of the selected window (`delete-window`). That is a zero. If there is more than one Emacs frame, deleting the sole remaining window on that frame deletes the frame as well. If the current frame is the only frame, it is not deleted.

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

C-x ^
: Make the selected window taller, at the expense of the other(s) (`enlarge-window`).

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

To delete a window, type *C-x 0* (`delete-window`). (That is a zero.) The space occupied by the deleted window is distributed among the other active windows (but not the minibuffer window, even if that is active at the time). Once a window is deleted, its attributes are forgotten; there is no automatic way to make another window of the same shape or showing the same buffer. 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 existing windows, use *C-x ^* (`enlarge-window`). It makes the currently selected window longer by one line or as many lines as a numeric argument specifies. With a negative argument, it makes the selected window smaller. *C-x }* (`enlarge-window-horizontally`) makes the selected window wider by the specified number of columns. The extra screen space given to a window comes from one of its neighbors, if that is possible; otherwise, all the competing windows are shrunk in the same proportion. If this makes some windows too small, those windows are deleted and their space is divided up. Minimum window size is specified by the variables `window-min-height` and `window-min-width`.

You can also resize windows within a frame by clicking the left mouse button on a modeline, and dragging.

Clicking the right button on a mode line pops up a menu of common window manager operations. This menu contains the following options:

Delete Window
: Remove the window above this modeline from the frame.

Delete Other Windows
: Delete all windows on the frame except for the one above this modeline.

Split Window
: Split the window above the mode line in half, creating another window.

Split Window Horizontally
: Split the window above the mode line in half horizontally, so that there will be two windows side-by-side.

Balance Windows
: Readjust the sizes of all windows on the frame until all windows have roughly the same number of lines.

18 World Scripts Support

If you compile XEmacs with mule option, it supports a wide variety of world scripts, including Latin script, as well as Arabic script, Simplified Chinese script (for mainland of China), Traditional Chinese script (for Taiwan and Hong-Kong), Greek script, Hebrew script, IPA symbols, Japanese scripts (Hiragana, Katakana and Kanji), Korean scripts (Hangul and Hanja) and Cyrillic script (for Beylorussian, Bulgarian, Russian, Serbian and Ukrainian). These features have been merged from the modified version of Emacs known as MULE (for "MULti-lingual Enhancement to GNU Emacs").

18.1 Introduction to world scripts

The users of these scripts have established many more-or-less standard coding systems for storing files. XEmacs translates between the internal character encoding and various other coding systems when reading and writing files, when exchanging data with subprocesses, and (in some cases) in the `C-q` command (see below).

The command `C-h h` (`view-hello-file`) displays the file '`etc/HELLO`', which shows how to say "hello" in many languages. This illustrates various scripts.

Keyboards, even in the countries where these character sets are used, generally don't have keys for all the characters in them. So XEmacs supports various *input methods*, typically one for each script or language, to make it convenient to type them.

The prefix key `C-x` ⟨RET⟩ is used for commands that pertain to world scripts, coding systems, and input methods.

18.2 Language Environments

All supported character sets are supported in XEmacs buffers if it is compile with mule; there is no need to select a particular language in order to display its characters in an XEmacs buffer. However, it is important to select a *language environment* in order to set various defaults. The language environment really represents a choice of preferred script (more or less) rather that a choice of language.

The language environment controls which coding systems to recognize when reading text (see Section 18.6 [Recognize Coding], page 139). This applies to files, incoming mail, netnews, and any other text you read into XEmacs. It may also specify the default coding system to use when you create a file. Each language environment also specifies a default input method.

The command to select a language environment is `M-x set-language-environment`. It makes no difference which buffer is current when you use this command, because the effects apply globally to the XEmacs session. The supported language environments include:

> Chinese-BIG5, Chinese-CNS, Chinese-GB, Cyrillic-ISO, English, Ethiopic, Greek, Japanese, Korean, Latin-1, Latin-2, Latin-3, Latin-4, Latin-5.

Some operating systems let you specify the language you are using by setting locale environment variables. XEmacs handles one common special case of this: if your locale name for character types contains the string '8859-*n*', XEmacs automatically selects the corresponding language environment.

To display information about the effects of a certain language environment *lang-env*, use the command *C-h L lang-env* ⟨RET⟩ (describe-language-environment). This tells you which languages this language environment is useful for, and lists the character sets, coding systems, and input methods that go with it. It also shows some sample text to illustrate scripts used in this language environment. By default, this command describes the chosen language environment.

18.3 Input Methods

An *input method* is a kind of character conversion designed specifically for interactive input. In XEmacs, typically each language has its own input method; sometimes several languages which use the same characters can share one input method. A few languages support several input methods.

The simplest kind of input method works by mapping ASCII letters into another alphabet. This is how the Greek and Russian input methods work.

A more powerful technique is composition: converting sequences of characters into one letter. Many European input methods use composition to produce a single non-ASCII letter from a sequence that consists of a letter followed by accent characters. For example, some methods convert the sequence *'a* into a single accented letter.

The input methods for syllabic scripts typically use mapping followed by composition. The input methods for Thai and Korean work this way. First, letters are mapped into symbols for particular sounds or tone marks; then, sequences of these which make up a whole syllable are mapped into one syllable sign.

Chinese and Japanese require more complex methods. In Chinese input methods, first you enter the phonetic spelling of a Chinese word (in input method chinese-py, among others), or a sequence of portions of the character (input methods chinese-4corner and chinese-sw, and others). Since one phonetic spelling typically corresponds to many different Chinese characters, you must select one of the alternatives using special XEmacs commands. Keys such as *C-f*, *C-b*, *C-n*, *C-p*, and digits have special definitions in this situation, used for selecting among the alternatives. ⟨TAB⟩ displays a buffer showing all the possibilities.

In Japanese input methods, first you input a whole word using phonetic spelling; then, after the word is in the buffer, XEmacs converts it into one or more characters using a large dictionary. One phonetic spelling corresponds to many differently written Japanese words, so you must select one of them; use *C-n* and *C-p* to cycle through the alternatives.

Sometimes it is useful to cut off input method processing so that the characters you have just entered will not combine with subsequent characters. For example, in input method latin-1-postfix, the sequence *e '* combines to form an 'e' with an accent. What if you want to enter them as separate characters?

One way is to type the accent twice; that is a special feature for entering the separate letter and accent. For example, *e ' '* gives you the two characters 'e''. Another way is to type another letter after the `e`—something that won't combine with that—and immediately delete it. For example, you could type *e e* ⟨DEL⟩ *'* to get separate '`e`' and '`'`'.

Another method, more general but not quite as easy to type, is to use *C-\ C-* between two characters to stop them from combining. This is the command *C-* (`toggle-input-method`) used twice.

C-\ C- is especially useful inside an incremental search, because stops waiting for more characters to combine, and starts searching for what you have already entered.

The variables `input-method-highlight-flag` and `input-method-verbose-flag` control how input methods explain what is happening. If `input-method-highlight-flag` is non-`nil`, the partial sequence is highlighted in the buffer. If `input-method-verbose-flag` is non-`nil`, the list of possible characters to type next is displayed in the echo area (but not when you are in the minibuffer).

18.4 Selecting an Input Method

C- Enable or disable use of the selected input method.

C-x ⟨RET⟩ *C-\ method* ⟨RET⟩
Select a new input method for the current buffer.

C-h I method ⟨RET⟩
C-h C-\ method ⟨RET⟩
Describe the input method *method* (`describe-input-method`). By default, it describes the current input method (if any).

M-x list-input-methods
Display a list of all the supported input methods.

To choose an input method for the current buffer, use *C-x* ⟨RET⟩ *C-* (`select-input-method`). This command reads the input method name with the minibuffer; the name normally starts with the language environment that it is meant to be used with. The variable `current-input-method` records which input method is selected.

Input methods use various sequences of ASCII characters to stand for non-ASCII characters. Sometimes it is useful to turn off the input method temporarily. To do this, type *C-* (`toggle-input-method`). To reenable the input method, type *C-* again.

If you type *C-* and you have not yet selected an input method, it prompts for you to specify one. This has the same effect as using *C-x* ⟨RET⟩ *C-* to specify an input method.

Selecting a language environment specifies a default input method for use in various buffers. When you have a default input method, you can select it in the current buffer by typing *C-*. The variable `default-input-method` specifies the default input method (`nil` means there is none).

Some input methods for alphabetic scripts work by (in effect) remapping the keyboard to emulate various keyboard layouts commonly used for those scripts. How to do this remapping properly depends on your actual keyboard layout. To specify which layout your keyboard has, use the command *M-x quail-set-keyboard-layout*.

To display a list of all the supported input methods, type `M-x list-input-methods`. The list gives information about each input method, including the string that stands for it in the mode line.

18.5 Coding Systems

Users of various languages have established many more-or-less standard coding systems for representing them. XEmacs does not use these coding systems internally; instead, it converts from various coding systems to its own system when reading data, and converts the internal coding system to other coding systems when writing data. Conversion is possible in reading or writing files, in sending or receiving from the terminal, and in exchanging data with subprocesses.

XEmacs assigns a name to each coding system. Most coding systems are used for one language, and the name of the coding system starts with the language name. Some coding systems are used for several languages; their names usually start with '`iso`'. There are also special coding systems `binary` and `no-conversion` which do not convert printing characters at all.

In addition to converting various representations of non-ASCII characters, a coding system can perform end-of-line conversion. XEmacs handles three different conventions for how to separate lines in a file: newline, carriage-return linefeed, and just carriage-return.

`C-h C` *coding* ⟨RET⟩
: Describe coding system *coding*.

`C-h C` ⟨RET⟩
: Describe the coding systems currently in use.

`M-x list-coding-systems`
: Display a list of all the supported coding systems.

The command `C-h C` (`describe-coding-system`) displays information about particular coding systems. You can specify a coding system name as argument; alternatively, with an empty argument, it describes the coding systems currently selected for various purposes, both in the current buffer and as the defaults, and the priority list for recognizing coding systems (see Section 18.6 [Recognize Coding], page 139).

To display a list of all the supported coding systems, type `M-x list-coding-systems`. The list gives information about each coding system, including the letter that stands for it in the mode line (see Section 1.3 [Mode Line], page 15).

Each of the coding systems that appear in this list—except for `binary`, which means no conversion of any kind—specifies how and whether to convert printing characters, but leaves the choice of end-of-line conversion to be decided based on the contents of each file. For example, if the file appears to use carriage-return linefeed between lines, that end-of-line conversion will be used.

Each of the listed coding systems has three variants which specify exactly what to do for end-of-line conversion:

`...-unix`
: Don't do any end-of-line conversion; assume the file uses newline to separate lines. (This is the convention normally used on Unix and GNU systems.)

`...-dos` Assume the file uses carriage-return linefeed to separate lines, and do the appropriate conversion. (This is the convention normally used on Microsoft systems.)

`...-mac` Assume the file uses carriage-return to separate lines, and do the appropriate conversion. (This is the convention normally used on the Macintosh system.)

These variant coding systems are omitted from the `list-coding-systems` display for brevity, since they are entirely predictable. For example, the coding system `iso-8859-1` has variants `iso-8859-1-unix`, `iso-8859-1-dos` and `iso-8859-1-mac`.

In contrast, the coding system `binary` specifies no character code conversion at all—none for non-Latin-1 byte values and none for end of line. This is useful for reading or writing binary files, tar files, and other files that must be examined verbatim.

The easiest way to edit a file with no conversion of any kind is with the *`M-x find-file-literally`* command. This uses `binary`, and also suppresses other XEmacs features that might convert the file contents before you see them. See Section 15.2 [Visiting], page 102.

The coding system `no-conversion` means that the file contains non-Latin-1 characters stored with the internal XEmacs encoding. It handles end-of-line conversion based on the data encountered, and has the usual three variants to specify the kind of end-of-line conversion.

18.6 Recognizing Coding Systems

Most of the time, XEmacs can recognize which coding system to use for any given file–once you have specified your preferences.

Some coding systems can be recognized or distinguished by which byte sequences appear in the data. However, there are coding systems that cannot be distinguished, not even potentially. For example, there is no way to distinguish between Latin-1 and Latin-2; they use the same byte values with different meanings.

XEmacs handles this situation by means of a priority list of coding systems. Whenever XEmacs reads a file, if you do not specify the coding system to use, XEmacs checks the data against each coding system, starting with the first in priority and working down the list, until it finds a coding system that fits the data. Then it converts the file contents assuming that they are represented in this coding system.

The priority list of coding systems depends on the selected language environment (see Section 18.2 [Language Environments], page 135). For example, if you use French, you probably want XEmacs to prefer Latin-1 to Latin-2; if you use Czech, you probably want Latin-2 to be preferred. This is one of the reasons to specify a language environment.

However, you can alter the priority list in detail with the command *`M-x prefer-coding-system`*. This command reads the name of a coding system from the minibuffer, and adds it to the front of the priority list, so that it is preferred to all others. If you use this command several times, each use adds one element to the front of the priority list.

Sometimes a file name indicates which coding system to use for the file. The variable `file-coding-system-alist` specifies this correspondence. There is a special function `modify-coding-`

`system-alist` for adding elements to this list. For example, to read and write all '`.txt`' using the coding system `china-iso-8bit`, you can execute this Lisp expression:

```
(modify-coding-system-alist 'file "\\.txt\\'" 'china-iso-8bit)
```

The first argument should be `file`, the second argument should be a regular expression that determines which files this applies to, and the third argument says which coding system to use for these files.

You can specify the coding system for a particular file using the '`-*-...-*-`' construct at the beginning of a file, or a local variables list at the end (see Section 29.2.5 [File Variables], page 266). You do this by defining a value for the "variable" named `coding`. XEmacs does not really have a variable `coding`; instead of setting a variable, it uses the specified coding system for the file. For example, '`-*-mode: C; coding: iso-8859-1;-*-`' specifies use of the iso-8859-1 coding system, as well as C mode.

Once XEmacs has chosen a coding system for a buffer, it stores that coding system in `buffer-file-coding-system` and uses that coding system, by default, for operations that write from this buffer into a file. This includes the commands `save-buffer` and `write-region`. If you want to write files from this buffer using a different coding system, you can specify a different coding system for the buffer using `set-buffer-file-coding-system` (see Section 18.7 [Specify Coding], page 140).

18.7 Specifying a Coding System

In cases where XEmacs does not automatically choose the right coding system, you can use these commands to specify one:

C-x ⟨RET⟩ *f coding* ⟨RET⟩
: Use coding system *coding* for the visited file in the current buffer.

C-x ⟨RET⟩ *c coding* ⟨RET⟩
: Specify coding system *coding* for the immediately following command.

C-x ⟨RET⟩ *k coding* ⟨RET⟩
: Use coding system *coding* for keyboard input.

C-x ⟨RET⟩ *t coding* ⟨RET⟩
: Use coding system *coding* for terminal output.

C-x ⟨RET⟩ *p coding* ⟨RET⟩
: Use coding system *coding* for subprocess input and output in the current buffer.

The command *C-x RET f* (`set-buffer-file-coding-system`) specifies the file coding system for the current buffer—in other words, which coding system to use when saving or rereading the visited file. You specify which coding system using the minibuffer. Since this command applies to a file you have already visited, it affects only the way the file is saved.

Another way to specify the coding system for a file is when you visit the file. First use the command *C-x* ⟨RET⟩ *c* (`universal-coding-system-argument`); this command uses the minibuffer to read a coding system name. After you exit the minibuffer, the specified coding system is used for *the immediately following command*.

So if the immediately following command is `C-x C-f`, for example, it reads the file using that coding system (and records the coding system for when the file is saved). Or if the immediately following command is `C-x C-w`, it writes the file using that coding system. Other file commands affected by a specified coding system include `C-x C-i` and `C-x C-v`, as well as the other-window variants of `C-x C-f`.

In addition, if you run some file input commands with the precedent `C-u`, you can specify coding system to read from minibuffer. So if the immediately following command is `C-x C-f`, for example, it reads the file using that coding system (and records the coding system for when the file is saved). Other file commands affected by a specified coding system include `C-x C-i` and `C-x C-v`, as well as the other-window variants of `C-x C-f`.

The variable `default-buffer-file-coding-system` specifies the choice of coding system to use when you create a new file. It applies when you find a new file, and when you create a buffer and then save it in a file. Selecting a language environment typically sets this variable to a good choice of default coding system for that language environment.

The command `C-x` ⟨RET⟩ `t` (`set-terminal-coding-system`) specifies the coding system for terminal output. If you specify a character code for terminal output, all characters output to the terminal are translated into that coding system.

This feature is useful for certain character-only terminals built to support specific languages or character sets—for example, European terminals that support one of the ISO Latin character sets.

By default, output to the terminal is not translated at all.

The command `C-x` ⟨RET⟩ `k` (`set-keyboard-coding-system`) specifies the coding system for keyboard input. Character-code translation of keyboard input is useful for terminals with keys that send non-ASCII graphic characters—for example, some terminals designed for ISO Latin-1 or subsets of it.

By default, keyboard input is not translated at all.

There is a similarity between using a coding system translation for keyboard input, and using an input method: both define sequences of keyboard input that translate into single characters. However, input methods are designed to be convenient for interactive use by humans, and the sequences that are translated are typically sequences of ASCII printing characters. Coding systems typically translate sequences of non-graphic characters.

The command `C-x` ⟨RET⟩ `p` (`set-buffer-process-coding-system`) specifies the coding system for input and output to a subprocess. This command applies to the current buffer; normally, each subprocess has its own buffer, and thus you can use this command to specify translation to and from a particular subprocess by giving the command in the corresponding buffer.

By default, process input and output are not translated at all.

The variable `file-name-coding-system` specifies a coding system to use for encoding file names. If you set the variable to a coding system name (as a Lisp symbol or a string), XEmacs encodes file names using that coding system for all file operations. This makes it possible to use non-Latin-1 characters in file names—or, at least, those non-Latin-1 characters which the specified coding system can encode. By default, this variable is `nil`, which implies that you cannot use non-Latin-1 characters in file names.

19 Major Modes

Emacs has many different *major modes*, each of which customizes Emacs for editing text of a particular sort. The major modes are mutually exclusive; at any time, each buffer has one major mode. The mode line normally contains the name of the current major mode in parentheses. See Section 1.3 [Mode Line], page 15.

The least specialized major mode is called *Fundamental mode*. This mode has no mode-specific redefinitions or variable settings. 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. ⟨TAB⟩, ⟨DEL⟩, and ⟨LFD⟩ are changed frequently. In addition, commands which handle comments use the mode to determine how to delimit comments. Many major modes redefine the syntactical properties of characters appearing in the buffer. See Section 29.5 [Syntax], page 274.

The major modes fall into three major groups. Lisp mode (which has several variants), C mode, and Muddle mode are for specific programming languages. Text mode, Nroff mode, TeX mode, and Outline mode are for editing English text. The remaining major modes are not intended for use on users' files; they are used in buffers created by Emacs for specific purposes and include Dired mode for buffers made by Dired (see Section 15.9 [Dired], page 120), Mail mode for buffers made by `C-x m` (see Chapter 26 [Sending Mail], page 211), and Shell mode for buffers used for communicating with an inferior shell process (see Section 28.2.2 [Interactive Shell], page 249).

Most programming language major modes specify that only blank lines separate paragraphs. This is so that the paragraph commands remain useful. See Section 21.4 [Paragraphs], page 158. 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 145.

19.1 Choosing Major Modes

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

Use a `M-x` command to explicitly select a new major mode. Add `-mode` to the name of a major mode to get the name of a command to select that mode. For example, to enter Lisp mode, execute `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 variable `auto-mode-alist` controls the correspondence between file names and major mode. 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)`. 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 non-blank line of the file. The mode name should appear in this line both preceded and followed by '`-*-`'. Other text may appear on the line as well. For example,

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

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

Another format of mode specification is:

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

which allows other things besides the major mode name to be specified. However, Emacs does not look for anything except the mode name.

The major mode can also be specified in a local variables list. See Section 29.2.5 [File Variables], page 266.

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*, Emacs uses the major mode specified by the variable `default-major-mode`. Normally this value is the symbol `fundamental-mode`, which specifies Fundamental mode. If `default-major-mode` is `nil`, the major mode is taken from the previously selected buffer.

20 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 non-blank 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 details differ.

Use the ⟨TAB⟩ command to indent a line whatever the language. Each major mode defines this command to perform indentation appropriate for the particular language. In Lisp mode, ⟨TAB⟩ aligns a 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, type *Meta-m* (`back-to-indentation`). This command, given anywhere on a line, positions point at the first non-blank character on the line.

To insert an indented line before the current line, type *C-a C-o* ⟨TAB⟩. To make an indented line after the current line, use *C-e* ⟨LFD⟩.

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

To join two lines cleanly, use the *Meta-^* (`delete-indentation`) command to delete the indentation at the front of the current line, and the line boundary as well. Empty spaces are replaced by a single space, or by no space if at the beginning of a line, before a close parenthesis, or after an open parenthesis. To delete just the indentation of a line, go to the beginning of the line and use *Meta-* (`delete-horizontal-space`), which deletes all spaces and tabs around the cursor.

There are also commands for changing the indentation of several lines at once. *Control-Meta-* (`indent-region`) gives each line which begins in the region the "usual" indentation by invoking ⟨TAB⟩ at the beginning of the line. A numeric argument specifies the column to indent to. Each line is shifted left or right so that its first non-blank character appears in that column. *C-x* ⟨TAB⟩ (`indent-rigidly`) moves all 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 non-empty line.) It inserts whitespace at point, moving point, until it is underneath an indentation point in the previous line. An indentation point is the end of a sequence of whitespace or the end of the line. If point is farther right than any indentation point in the previous line, the whitespace before point is deleted and the first indentation point then applicable is used. If no indentation point is applicable even then, `tab-to-tab-stop` is run (see next section).

`indent-relative` is the definition of ⟨TAB⟩ in Indented Text mode. See Chapter 21 [Text], page 149.

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. Even if you are not in Text mode, this function is associated with *M-i* anyway.

You can arbitrarily set the tab stops used by *M-i*. They are stored as a list of column-numbers in increasing order in the variable `tab-stop-list`.

The convenient way to set the tab stops is using *M-x edit-tab-stops*, which creates and selects a buffer containing a description of the tab stop settings. You can edit this buffer to specify different tab stops, and then type *C-c C-c* to make those new tab stops take effect. In the tab stop buffer, *C-c C-c* runs the function `edit-tab-stops-note-changes` rather than the default `save-buffer`. `edit-tab-stops` records which buffer was current when you invoked it, and stores the tab stops in that buffer. Normally all buffers share the same tab stops and changing them in one buffer affects all. If you make `tab-stop-list` local in one buffer `edit-tab-stops` in that buffer edits only the local settings.

Below is the text representing 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 help you see where the colons are and tell you what to do.

Note that the tab stops that control `tab-to-tab-stop` have nothing to do with displaying tab characters in the buffer. See Section 12.4 [Display Vars], page 83, 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 29.2.4 [Locals], page 264.

There are also commands to convert tabs to spaces or vice versa, always preserving the columns of all non-blank text. *`M-x tabify`* scans the region for sequences of spaces, and converts sequences of at least three spaces to tabs if that is possible without changing indentation. *`M-x untabify`* changes all tabs in the region to corresponding 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. In this sense of the word any file that you edit with Emacs is text. 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 and stylistic conventions that editor commands should support or use to advantage: conventions involving words, sentences, paragraphs, and capital letters. This chapter describes Emacs commands for all these things. There are also commands for *filling*, or rearranging paragraphs into lines of approximately equal length. The commands for moving over and killing words, sentences, and paragraphs, while intended primarily for editing text, are also often useful for editing programs.

Emacs has several major modes for editing human language text. If a file contains plain text, 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 can use TeX mode; for input to nroff, Nroff mode.

21.1 Text Mode

You should use Text mode—rather than Fundamental or Lisp mode—to edit files of text in a human language. 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 146). Features concerned with comments in programs are turned off unless they are 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 145), and makes Auto Fill indent the lines it creates. As a result, a line made by Auto Filling, or by ⟨LFD⟩, is normally indented just like the previous line. Use `M-x indented-text-mode` to select this mode.

Entering Text mode or Indented Text mode calls the value of the variable `text-mode-hook` with no arguments, if that value exists and is not `nil`. This value is also called when modes related to Text mode are entered; this includes Nroff mode, TeX mode, Outline mode, and Mail mode. Your hook can look at the value of `major-mode` to see which of these modes is actually being entered.

Two modes similar to Text mode are of use for editing text that is to be passed through a text formatter before achieving its final readable form.

21.1.1 Nroff Mode

Nroff mode is a mode like Text mode but modified to handle nroff commands present in the text. Invoke `M-x nroff-mode` to enter this mode. Nroff mode differs from Text mode in only a few ways.

All nroff command lines are considered paragraph separators, so that filling never garbles the nroff commands. Pages are separated by '`.bp`' commands. Comments start with backslash-doublequote. There are also three special commands that are not available 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 Electric Nroff newline mode. This is a minor mode that you can turn on or off with *M-x electric-nroff-mode* (see Section 29.1 [Minor Modes], page 257). When the mode is on and you use ⟨RET⟩ to end a line containing an nroff command that opens a kind of grouping, Emacs automatically inserts the matching nroff command to close that grouping on the following line. For example, if you are at the beginning of a line and type *.(b* ⟨RET⟩, the matching command '`.)b`' will be inserted on a new line following point.

Entering Nroff mode calls the value of the variable `text-mode-hook` with no arguments, if that value exists and is not `nil`; then it does the same with the variable `nroff-mode-hook`.

21.1.2 TEX Mode

TEX is a powerful text formatter written by Donald Knuth; like GNU Emacs, it is free. LaTEX is a simplified input format for TEX, implemented by TEX macros. It is part of TEX.

Emacs has a special TEX mode for editing TEX input files. It provides facilities for checking the balance of delimiters and for invoking TEX on all or part of the file.

TEX mode has two variants, Plain TEX mode and LaTEX mode, which are two distinct major modes that differ only slightly. These modes are designed for editing the two different input formats. The command *M-x tex-mode* looks at the contents of a buffer to determine whether it appears to be LaTEX input or not; it then selects the appropriate mode. If it can't tell which is right (e.g., the buffer is empty), the variable `tex-default-mode` controls which mode is used.

The commands *M-x plain-tex-mode* and *M-x latex-mode* explicitly select one of the variants of TEX mode. Use these commands when *M-x tex-mode* does not guess right.

TEX for Unix systems can be obtained from the University of Washington for a distribution fee.

To order a full distribution, send $140.00 for a 1/2 inch 9-track tape, $165.00 for two 4-track 1/4 inch cartridge tapes (foreign sites $150.00, for 1/2 inch, $175.00 for 1/4 inch, to cover the extra postage) payable to the University of Washington to:

The Director
Northwest Computer Support Group, DW-10
University of Washington
Seattle, Washington 98195

Purchase orders are acceptable, but there is an extra charge of $10.00 to pay for processing charges. (The total cost comes to $150 for domestic sites, $175 for foreign sites).

The normal distribution is a tar tape, blocked 20, 1600 bpi, on an industry standard 2400 foot half-inch reel. The physical format for the 1/4 inch streamer cartridges uses QIC-11, 8000 bpi, 4-track serpentine recording for the SUN. Also, SystemV tapes can be written in cpio format, blocked 5120 bytes, ASCII headers.

21.1.2.1 TeX Editing Commands

Here are the special commands provided in TeX mode for editing the text of the file.

`"` Insert, according to context, either '``' or '"' or '''' (`TeX-insert-quote`).

⟨LFD⟩ Insert a paragraph break (two newlines) and check the previous paragraph for unbalanced braces or dollar signs (`tex-terminate-paragraph`).

M-x validate-tex-buffer
Check each paragraph in the buffer for unbalanced braces or dollar signs.

C-c { Insert '`{}`' and position point between them (`tex-insert-braces`).

C-c } Move forward past the next unmatched close brace (`up-list`).

C-c C-e Close a block for LaTeX (`tex-close-latex-block`).

In TeX, the character '"' is not normally used; you use '``' to start a quotation and '''' to end one. TeX mode defines the key " to insert '``' after whitespace or an open brace, '"' after a backslash, or '''' otherwise. This is done by the command `tex-insert-quote`. If you need the character '"' itself in unusual contexts, use *C-q* to insert it. Also, " with a numeric argument always inserts that number of '"' characters.

In TeX mode, '`$`' has a special syntax code which attempts to understand the way TeX math mode delimiters match. When you insert a '`$`' that is meant to exit math mode, the position of the matching '`$`' that entered math mode is displayed for a second. This is the same feature that displays the open brace that matches a close brace that is inserted. However, there is no way to tell whether a '`$`' enters math mode or leaves it; so when you insert a '`$`' that enters math mode, the previous '`$`' position is shown as if it were a match, even though they are actually unrelated.

If you prefer to keep braces balanced at all times, you can use *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-buffer* checks the entire buffer, paragraph by paragraph. When it finds a paragraph that contains a mismatch, it displays point at the beginning of the paragraph for a few seconds and pushes a mark at that spot. Scanning continues until the whole buffer has been checked or until you type another key. The positions of the last several paragraphs with mismatches can be found in the mark ring (see Section 9.1.4 [Mark Ring], page 65).

Note that square brackets and parentheses, not just braces, are matched in TEX mode. This is wrong if you want to check TEX syntax. However, parentheses and square brackets are likely to be used in text as matching delimiters and it is useful for the various motion commands and automatic match display to work with them.

In LaTEX input, '`\begin`' and '`\end`' commands must balance. After you insert a '`\begin`', use *C-c C-f* (`tex-close-latex-block`) to insert automatically a matching '`\end`' (on a new line following the '`\begin`'). A blank line is inserted between the two, and point is left there.

21.1.2.2 TEX Printing Commands

You can invoke TEX as an inferior of Emacs on either the entire contents of the buffer or just a region at a time. Running TEX in this way on just one chapter is a good way to see what your changes look like without taking the time to format the entire file.

C-c C-r Invoke TEX on the current region, plus the buffer's header (`tex-region`).

C-c C-b Invoke TEX on the entire current buffer (`tex-buffer`).

C-c C-l Recenter the window showing output from the inferior TEX so that the last line can be seen (`tex-recenter-output-buffer`).

C-c C-k Kill the inferior TEX (`tex-kill-job`).

C-c C-p Print the output from the last *C-c C-r* or *C-c C-b* command (`tex-print`).

C-c C-q Show the printer queue (`tex-show-print-queue`).

You can pass the current buffer through an inferior TEX using *C-c C-b* (`tex-buffer`). The formatted output appears in a file in '`/tmp`'; to print it, type *C-c C-p* (`tex-print`). Afterward use *C-c C-q* (`tex-show-print-queue`) to view the progress of your output towards being printed.

The console output from TEX, including any error messages, 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 28.2.2 [Interactive Shell], page 249). 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 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 them, 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, insert two special strings in the file: '`%**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 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, so you don't need to do anything special to identify the header.

When you enter either kind of TeX mode, Emacs calls with no arguments the value of the variable `text-mode-hook`, if that value exists and is not `nil`. Emacs then calls the variable `TeX-mode-hook` and either `plain-TeX-mode-hook` or `LaTeX-mode-hook` under the same conditions.

21.1.3 Outline Mode

Outline mode is a major mode similar to Text mode but intended for editing outlines. It allows you to make parts of the text temporarily invisible so that you can see just the overall structure of the outline. Type *M-x outline-mode* to turn on Outline mode in the current buffer.

When you enter Outline mode, Emacs calls with no arguments the value of the variable `text-mode-hook`, if that value exists and is not `nil`; then it does the same with the variable `outline-mode-hook`.

When a line is invisible in outline mode, it does not appear on the screen. The screen appears exactly as if the invisible line were deleted, except that an ellipsis (three periods in a row) appears at the end of the previous visible line (only one ellipsis no matter how many invisible lines follow).

All editing commands treat the text of the invisible line as part of the previous visible line. For example, *C-n* moves onto the next visible line. Killing an entire visible line, including its terminating newline, really kills all the following invisible lines as well; yanking everything back yanks the invisible lines and they remain invisible.

21.1.3.1 Format of Outlines

Outline mode assumes that the lines in the buffer are of two types: *heading lines* and *body lines*. A heading line represents a topic in the outline. Heading lines start with one or more stars; the number of stars determines the depth of the heading in the outline structure. Thus, a heading line with one star is a major topic; all the heading lines with two stars between it and the next one-star heading are its subtopics; and so on. Any line that is not a heading line is a body line. Body lines belong to the preceding heading line. Here is an example:

```
* Food

This is the body,
which says something about the topic of food.

** Delicious Food

This is the body of the second-level header.

** Distasteful Food

This could have
a body too, with
```

```
several lines.

*** Dormitory Food

* Shelter

A second first-level topic with its header line.
```

A heading line together with all following body lines is called collectively an *entry*. A heading line together with all following deeper heading lines and their body lines is called a *subtree*.

You can customize the criterion for distinguishing heading lines by setting the variable `outline-regexp`. Any line whose beginning has a match for this regexp is considered a heading line. Matches that start within a line (not at the beginning) do not count. The length of the matching text determines the level of the heading; longer matches make a more deeply nested level. Thus, for example, if a text formatter has commands '`@chapter`', '`@section`' and '`@subsection`' to divide the document into chapters and sections, you can 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 the same length, 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. Saving does not change the visibility status of a line inside Emacs.

21.1.3.2 Outline Motion Commands

Some special commands in Outline mode move backward and forward to heading lines.

C-c C-n Move point to the next visible heading line (`outline-next-visible-heading`).

C-c C-p Move point to the previous visible heading line (`outline-previous-visible-heading`).

C-c C-f Move point to the next visible heading line at the same level as the one point is on (`outline-forward-same-level`).

C-c C-b Move point to the previous visible heading line at the same level (`outline-backward-same-level`).

C-c C-u Move point up to a lower-level (more inclusive) visible heading line (`outline-up-heading`).

C-c C-n (`next-visible-heading`) moves down to the next heading line. *C-c C-p* (`previous-visible-heading`) moves similarly backward. Both accept numeric arguments as repeat counts. The names emphasize that invisible headings are skipped, but this is not really a special feature. All editing commands that look for lines ignore the invisible lines automatically.

More advanced motion commands understand the levels of headings. The commands *C-c C-f* (`outline-forward-same-level`) and *C-c C-b* (`outline-backward-same-level`) move from one heading line to another visible heading at the same depth in the outline. *C-c C-u* (`outline-up-heading`) moves backward to another heading that is less deeply nested.

21.1.3.3 Outline Visibility Commands

The other special commands of outline mode are used to make lines visible or invisible. Their names all start with `hide` or `show`. Most of them exist as 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-d
: 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 should 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, one that hides the bodies and one that makes 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.

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. You can think of these commands 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`. The result is no visible indication of the presence of invisible lines.

21.2 Words

Emacs has commands for moving over or operating on words. By convention, the keys for them are all `Meta-` characters.

`M-f` Move forward over a word (`forward-word`).

`M-b` Move backward over a word (`backward-word`).

`M-d` Kill up to the end of a word (`kill-word`).

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

`M-@` Mark the end of the next word (`mark-word`).

`M-t` Transpose two words; drag a word forward or backward across other words (`transpose-words`).

Notice how these keys form a series that parallels the character-based `C-f`, `C-b`, `C-d`, `C-t` and `<DEL>`. `M-@` is related to `C-@`, which is an alias for `C-<SPC>`.

The commands `Meta-f` (`forward-word`) and `Meta-b` (`backward-word`) move forward and backward over words. They are analogous to `Control-f` and `Control-b`, which move over single characters. Like their `Control-` analogues, `Meta-f` and `Meta-b` move several words if given an argument. `Meta-f` with a negative argument moves backward, and `Meta-b` with a negative argument moves forward. Forward motion stops after the last letter of the word, while backward motion stops before the first letter.

`Meta-d` (`kill-word`) kills the word after point. To be precise, it kills everything from point to the place `Meta-f` would move to. Thus, if point is in the middle of a word, `Meta-d` kills just the part after point. If some punctuation comes between point and the next word, it is killed along with the word. (To kill only the next word but not the punctuation before it, simply type `Meta-f` to get to the end and kill the word backwards with `Meta-<DEL>`.) `Meta-d` takes arguments just like `Meta-f`.

`Meta-<DEL>` (`backward-kill-word`) kills the word before point. It kills everything from point back to where `Meta-b` would move to. If point is after the space in '`FOO, BAR`', then '`FOO, `' is killed. To kill just '`FOO`', type `Meta-b Meta-d` instead of `Meta-<DEL>`.

`Meta-t` (`transpose-words`) exchanges the word before or containing point with the following word. The delimiter characters between the words do not move. For example, transposing '`FOO, BAR`' results in '`BAR, FOO`' rather than '`BAR FOO,`'. See Section 14.2 [Transpose], page 97, for more on transposition and on arguments to transposition commands.

To operate on the next *n* words with an operation which applies between point and mark, you can either set the mark at point and then move over the words, or you can use the command *Meta-@* (`mark-word`) which does not move point but sets the mark where *Meta-f* would move to. It can be given arguments just like *Meta-f*.

The word commands' understanding of syntax is completely controlled by the syntax table. For example, any character can be declared to be a word delimiter. See Section 29.5 [Syntax], page 274.

21.3 Sentences

The Emacs commands for manipulating sentences and paragraphs are mostly on *Meta-* keys, and therefore are like the word-handling commands.

M-a Move back to the beginning of the sentence (`backward-sentence`).

M-e Move forward to the end of the sentence (`forward-sentence`).

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

C-x ⟨DEL⟩ Kill back to the beginning of the sentence (`backward-kill-sentence`).

The commands *Meta-a* and *Meta-e* (`backward-sentence` and `forward-sentence`) move to the beginning and end of the current sentence, respectively. They resemble *Control-a* and *Control-e*, which move to the beginning and end of a line. Unlike their counterparts, *Meta-a* and *Meta-e* move over successive sentences if repeated or given numeric arguments. Emacs assumes 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.

M-a and *M-e* have a corresponding kill command, just like *C-a* and *C-e* have *C-k*. The command is *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 repeat counts.

There is a special command, *C-x* ⟨DEL⟩ (`backward-kill-sentence`), for killing back to the beginning of a sentence, which 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 89.

21.4 Paragraphs

The Emacs commands for manipulating paragraphs are also *Meta-* keys.

M-[
: Move back to previous paragraph beginning (`backward-paragraph`).

M-]
: Move forward to next paragraph end (`forward-paragraph`).

M-h
: Put point and mark around this or next paragraph (`mark-paragraph`).

Meta-[moves to the beginning of the current or previous paragraph, while *Meta-]* moves to the end of the current or next paragraph. Blank lines and text formatter command lines separate paragraphs and are not part of any paragraph. 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. As a result, the paragraph commands continue to be useful even though there are no paragraphs per se.

When there is a fill prefix, paragraphs are delimited by all lines which don't start with the fill prefix. See Section 21.6 [Filling], page 159.

To operate on a paragraph, you can use the command *Meta-h* (`mark-paragraph`) to set the region around it. This command puts point at the beginning and mark at the end of the paragraph point was in. If point is between paragraphs (in a run of blank lines or at a boundary), the paragraph following point is surrounded by point and mark. If there are blank lines preceding the first line of the paragraph, one of the 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 matches any line that either starts or separates paragraphs. The value of `paragraph-separate` is another regexp that matches only lines that separate paragraphs without being part of any paragraph. Lines that start a new paragraph and are contained in it must match both regexps. For example, normally `paragraph-start` is `"^[ \t\n\f]"` and `paragraph-separate` is `"^[ \t\f]*$"`.

Normally it is desirable for page boundaries to separate paragraphs. The default values of these variables recognize the usual separator for pages.

21.5 Pages

Files are often thought of as divided into *pages* by the *formfeed* character (ASCII Control-L, octal code 014). For example, if a file is printed on a line printer, each "page" of the file starts on a new page of paper. Emacs treats a page-separator character just like any other character. It can be inserted with *C-q C-l* or deleted with ⟨DEL⟩. You are free to paginate your file or not. However, since pages are often meaningful divisions of the file, commands are provided to move over them and operate on them.

C-x [
: Move point to previous page boundary (`backward-page`).

C-x]
: Move point to next page boundary (`forward-page`).

C-x C-p Put point and mark around this page (or another page) (`mark-page`).

C-x l Count the lines in this page (`count-lines-page`).

The *C-x [* (`backward-page`) command moves point to immediately after the previous page delimiter. If point is already right after a page delimiter, the command 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). You can follow this command by *C-w* to kill a page you want to move elsewhere. If you insert the page after a page delimiter, at a place where *C-x]* or *C-x [* would take you, 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`) can help you decide where to break a page in two. It prints the total number of lines in the current page in the echo area, then divides the lines into those preceding the current line and those following it, for example

```
Page has 96 (72+25) lines
```

Notice that the sum is off by one; this is correct if point is not at the beginning of a line.

The variable `page-delimiter` should have as its value a regexp that matches the beginning of a line that separates pages. This defines where pages begin. The normal value of this variable is `"^\f"`, which matches a formfeed character at the beginning of a line.

21.6 Filling Text

If you use Auto Fill mode, Emacs *fills* text (breaks it up into lines that fit in a specified width) as you insert it. When you alter existing text it is often no longer be properly filled afterwards and you can use explicit commands for filling.

21.6.1 Auto Fill Mode

Auto Fill mode is a minor mode in which lines are broken automatically when they become too wide. Breaking happens only when you type a ⟨SPC⟩ or ⟨RET⟩.

M-x auto-fill-mode
Enable or disable Auto Fill mode.

⟨SPC⟩
⟨RET⟩ In Auto Fill mode, break lines when appropriate.

M-x auto-fill-mode turns Auto Fill mode on if it was off, or off if it was on. With a positive numeric argument the command always turns Auto Fill mode on, and with a negative argument it always turns it off. The presence of the word '`Fill`' in the mode line, inside the parentheses, indicates that Auto Fill mode is in effect. Auto Fill mode is a minor mode; you can turn it on or off for each buffer individually. See Section 29.1 [Minor Modes], page 257.

In Auto Fill mode, lines are broken automatically at spaces when they get longer than desired. Line breaking and rearrangement takes place only when you type ⟨SPC⟩ or ⟨RET⟩. 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). *C-o* inserts a newline without line breaking.

Auto Fill mode works well with Lisp mode: when it makes a new line in Lisp mode, it indents that line with ⟨TAB⟩. If a line ending in a Lisp 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 22.6 [Comments], page 172).

Auto Fill mode does not refill entire paragraphs. It can break lines but cannot merge lines. 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 using an explicit fill commands.

Many users like Auto Fill mode and want to use it in all text files. The section on init files explains how you can arrange this permanently for yourself. See Section 29.6 [Init File], page 277.

21.6.2 Explicit Fill Commands

M-q Fill current paragraph (`fill-paragraph`).

M-g Fill each paragraph in the region (`fill-region`).

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

M-x fill-region-as-paragraph
Fill the region, considering it as one paragraph.

M-s Center a line.

To refill a paragraph, use the command *Meta-q* (`fill-paragraph`). It causes the paragraph containing point, or the one after point if point is between paragraphs, to be refilled. All line breaks are removed, and new ones are inserted where necessary. *M-q* can be undone with *C-_*. See Chapter 5 [Undo], page 49.

To refill many paragraphs, use *M-g* (`fill-region`), which divides the region into paragraphs and fills each of them.

Meta-q and *Meta-g* use the same criteria as *Meta-h* for finding paragraph boundaries (see Section 21.4 [Paragraphs], page 158). For more control, you can use *M-x fill-region-as-paragraph*, which refills everything between point and mark. This command recognizes only blank lines as paragraph separators.

A numeric argument to *M-g* or *M-q* causes it to *justify* the text as well as filling it. Extra spaces are inserted to make the right margin line up exactly at the fill column. To remove the extra spaces, use *M-q* or *M-g* with no argument.

The variable `auto-fill-inhibit-regexp` takes as a value a regexp to match lines that should not be auto-filled.

The command *Meta-s* (`center-line`) centers the current line within the current fill column. With an argument, it centers several lines individually and moves past them.

The maximum line width for filling is in the variable `fill-column`. Altering the value of `fill-column` makes it local to the current buffer; until then, the default value—initially 70—is in effect. See Section 29.2.4 [Locals], page 264.

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 number as the new fill column.

21.6.3 The Fill Prefix

To fill a paragraph in which each line starts with a special marker (which might be a few spaces, giving an indented paragraph), use the *fill prefix* feature. The fill prefix is a string which is not included in filling. Emacs expects every line to start with a fill prefix.

C-x .
: Set the fill prefix (`set-fill-prefix`).

M-q
: Fill a paragraph using current fill prefix (`fill-paragraph`).

M-x fill-individual-paragraphs
: Fill the region, considering each change of indentation as starting a new paragraph.

To specify a fill prefix, move to a line that starts with the desired prefix, put point at the end of the prefix, and give the command *C-x .* (`set-fill-prefix`). That's a period after the *C-x*. To turn off the fill prefix, specify an empty prefix: type *C-x .* with point at the beginning of a line.

When a fill prefix is in effect, the fill commands remove the fill prefix from each line before filling and insert it on each line after filling. Auto Fill mode also inserts the fill prefix inserted on new lines it creates. Lines that do not start with the fill prefix are considered to start paragraphs, both in *M-q* and the paragraph commands; this is just right if you are using paragraphs with hanging indentation (every line indented except the first one). Lines which are blank or indented once the prefix is removed also separate or start paragraphs; this is what you want if you are writing multi-paragraph comments with a comment delimiter on each line.

The fill prefix is stored in the variable `fill-prefix`. Its value is a string, or `nil` when there is no fill prefix. This is a per-buffer variable; altering the variable affects only the current buffer, but there is a default value which you can change as well. See Section 29.2.4 [Locals], page 264.

Another way to use fill prefixes is through *M-x fill-individual-paragraphs*. This function divides the region into groups of consecutive lines with the same amount and kind of indentation and fills each group as a paragraph, using its indentation as a fill prefix.

21.7 Case Conversion Commands

Emacs has commands for converting either a single word or any arbitrary range of text to upper case or to lower case.

M-l Convert following word to lower case (`downcase-word`).

M-u Convert following word to upper case (`upcase-word`).

M-c Capitalize the following word (`capitalize-word`).

C-x C-l Convert region to lower case (`downcase-region`).

C-x C-u Convert region to upper case (`upcase-region`).

The word conversion commands are used most frequently. *Meta-l* (`downcase-word`) converts the word after point to lower case, moving past it. Thus, repeating *Meta-l* converts successive words. *Meta-u* (`upcase-word`) converts to all capitals instead, while *Meta-c* (`capitalize-word`) puts the first letter of the word into upper case and the rest into lower case. The word conversion 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: you can move through the text using *M-l*, *M-u*, or *M-c* on each word as appropriate, occasionally using *M-f* instead to skip a word.

When given a negative argument, the word case conversion commands apply to the appropriate number of words before point, but do not move point. This is convenient when you have just typed a word in the wrong case: you can give the case conversion command and continue typing.

If a word case conversion command is given in the middle of a word, it applies only to the part of the word which follows point. This is just like what *Meta-d* (`kill-word`) does. With a negative argument, case conversion applies only to the part of the word before point.

The other case conversion commands are *C-x C-u* (`upcase-region`) and *C-x C-l* (`downcase-region`), which convert everything between point and mark to the specified case. Point and mark do not move.

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 balanced expressions (*defuns*, in Lisp; functions, in C).
- Show how parentheses balance (see Section 22.5 [Matching], page 171).
- Insert, kill, or align comments (see Section 22.6 [Comments], page 172).
- Follow the usual indentation conventions of the language (see Section 22.4 [Grinding], page 166).

The commands available for words, sentences, and paragraphs are useful in editing code even though their canonical application is for editing human language text. Most symbols contain words (see Section 21.2 [Words], page 156); sentences can be found in strings and comments (see Section 21.3 [Sentences], page 157). Paragraphs per se are not present in code, but the paragraph commands are useful anyway, because Lisp mode and C mode define paragraphs to begin and end at blank lines (see Section 21.4 [Paragraphs], page 158). Judicious use of blank lines to make the program clearer also provides 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 82). 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 has several major modes for the programming languages Lisp, Scheme (a variant of Lisp), C, Fortran, and Muddle. Ideally, a major mode should be implemented for each programming language 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 variants of Lisp mode, which differ in the way they interface to Lisp execution. See Section 23.2 [Lisp Modes], page 190.

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. In these modes, use `C-b C-d` to delete a tab character before point.

Programming language modes define paragraphs to be separated only by blank lines, so that the paragraph commands remain useful. Auto Fill mode, if enabled in a programming language major mode, indents the new lines which it creates.

Turning on a major mode calls a user-supplied function called the *mode hook*, which is the value of a Lisp variable. For example, turning on C mode calls the value of the variable `c-mode-hook` if that value exists and is non-`nil`. Mode hook variables for other programming language modes include `lisp-mode-hook`, `emacs-lisp-mode-hook`, `lisp-interaction-mode-hook`, `scheme-mode-hook`, and `muddle-mode-hook`. The mode hook function receives no arguments.

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 English).

The commands fall into two classes. Some commands deal only with *lists* (parenthetical groupings). They see nothing except parentheses, brackets, braces (depending on what 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 term for a symbolic expression in Lisp. In Emacs, the notion of 'sexp' is not limited to Lisp. It refers to an expression in the 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 an sexp, even though it *is* a C expression; it recognizes '`foo`' as one sexp and '`bar`' as another, with the '`+`' as punctuation between them. This is a fundamental ambiguity: both '`foo + bar`' and '`foo`' are legitimate choices for the sexp to move over if point is at the '`f`'. Note that '`(foo + bar)`' is a sexp in C mode.

Some languages have obscure forms of syntax for expressions that nobody has bothered to make Emacs understand properly.

C-M-f Move forward over an sexp (`forward-sexp`).

C-M-b Move backward over an 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 an sexp, use `C-M-f` (`forward-sexp`). If the first significant character after point is an opening delimiter ('(' in Lisp; '(', '[', or '{' in C), `C-M-f` moves past the matching closing delimiter. If the character begins a symbol, string, or number, `C-M-f` moves over that. If the character after point is a closing delimiter, `C-M-f` just moves past it. (This last is not really moving across an sexp; it is an exception which is included in the definition of `C-M-f` because it is as useful a behavior as anyone can think of for that situation.)

The command `C-M-b` (`backward-sexp`) moves backward over a sexp. The detailed rules are like those above for `C-M-f`, but with directions reversed. If there are any prefix characters (single quote, back quote, and comma, in Lisp) preceding the sexp, `C-M-b` moves back over them as well.

`C-M-f` or `C-M-b` with an argument repeats that operation the specified number of times; with a negative argument, it moves in the opposite direction.

In languages such as C where the comment-terminator can be recognized, the sexp commands move across comments as if they were whitespace. In Lisp and other languages where comments run until the end of a line, it is very difficult to ignore comments when parsing backwards; therefore, in such languages the sexp commands treat the text of comments as if it were code.

Killing an 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*, `C-M-n` (`forward-list`) and `C-M-p` (`backward-list`), move over lists like the sexp commands but skip over any number of other kinds of sexps (symbols, strings, etc). In some situations, these commands are useful because 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 is 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.

`C-M-t` (`transpose-sexps`) 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 the 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 29.5 [Syntax], page 274.

22.3 Defuns

In Emacs, a parenthetical grouping at the top level in the buffer is called a *defun*. The name derives from the fact that most top-level lists in Lisp are instances of the special form `defun`, but Emacs calls any top-level parenthetical grouping counts a defun regardless of its contents or the programming language. 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`).

To operate on the current defun, use *C-M-h* (`mark-defun`) which puts point at the beginning and the mark at the end of the current or next defun. This is the easiest way to prepare for moving the defun to a different place. In C mode, *C-M-h* runs the function `mark-c-function`, which is almost the same as `mark-defun`, but which backs up over the argument declarations, function name, and returned data type so that the entire C function is inside the region.

To compile and evaluate the current defun, use *M-x compile-defun*. This function prints the results in the minibuffer. If you include an argument, it inserts the value in the current buffer after the defun.

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.

The original Emacs found defuns by moving upward a level of parentheses until there were no more levels to go up. This required scanning back to the beginning of the buffer for every function. To speed this up, 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; however, it mandates the convention described above.

22.4 Indentation for Programs

The best way to keep a program properly indented ("ground") is to use Emacs to re-indent it as you change the program. 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.

22.4.1 Basic Program Indentation Commands

⟨TAB⟩ Adjust indentation of current line.

⟨LFD⟩ Equivalent to ⟨RET⟩ followed by ⟨TAB⟩ (`newline-and-indent`).

The basic indentation command is ⟨TAB⟩, which gives the current line the correct indentation as determined from the previous lines. The function that ⟨TAB⟩ runs depends on the major mode; it is `lisp-indent-line` in Lisp mode, `c-indent-line` in C mode, etc. These functions understand different syntaxes for different languages, but they all do about the same thing. ⟨TAB⟩ in any programming language major mode inserts or deletes whitespace at the beginning of the current line, independent of where point is in the line. If point is inside the whitespace at the beginning of the line, ⟨TAB⟩ leaves it at the end of that whitespace; otherwise, ⟨TAB⟩ leaves point fixed with respect to the characters around it.

Use *C-q* ⟨TAB⟩ to insert a tab at point.

When entering a large amount of new code, use ⟨LFD⟩ (`newline-and-indent`), which is equivalent to a ⟨RET⟩ followed by a ⟨TAB⟩. ⟨LFD⟩ creates a blank line, 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 tend to follow it. This is the right behavior in cases where the standard result of ⟨TAB⟩ does not look good.

Remember that Emacs assumes that an open-parenthesis, open-brace, or other opening delimiter at the left margin (including the indentation routines) is the start of a function. You should therefore 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. See Section 22.3 [Defuns], page 166, for more information on this behavior.

22.4.2 Indenting Several Lines

Several commands are available to re-indent several lines of code which have been altered or moved to a different level in a list structure.

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`).

To re-indent the contents of a single list, position point before the beginning of it and type *C-M-q*. This key is bound to `indent-sexp` in Lisp mode, `indent-c-exp` in C mode, and bound to other suitable functions in other modes. The indentation of the line the sexp starts on is not changed; therefore, only the relative indentation within the list, and not its position, is changed. To correct the position as well, type a ⟨TAB⟩ before *C-M-q*.

If the relative indentation within a list is correct but the indentation of its beginning is not, go to the line on which the list begins and type *C-u* ⟨TAB⟩. When you give ⟨TAB⟩ a numeric argument, it moves all the lines in the group, starting on the current line, sideways the same amount that the current line moves. The command does not move lines that start inside strings, or C preprocessor lines when in C mode.

Another way to specify a 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.

22.4.3 Customizing Lisp Indentation

The indentation pattern for a Lisp expression can depend on the function called by the expression. For each Lisp function, you can choose among several predefined patterns of indentation, or define an arbitrary one with a Lisp program.

The standard pattern of indentation is as follows: the second line of the expression is indented under the first argument, if that is on the same line as the beginning of the expression; otherwise, the second line is indented underneath the function name. Each following line is indented under the previous line whose nesting depth is the same.

If the variable `lisp-indent-offset` is non-`nil`, it overrides the usual indentation pattern for the second line of an expression, so that such lines are always indented `lisp-indent-offset` more columns than the containing list.

Certain functions override the standard pattern. Functions whose names start with `def` always indent the second line by `lisp-body-indention` extra columns beyond the open-parenthesis starting the expression.

Individual functions can override the standard pattern in various ways, according to the `lisp-indent-function` property of the function name. (Note: `lisp-indent-function` was formerly called `lisp-indent-hook`). 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 first element 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 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.

22.4.4 Customizing C Indentation

Two variables control which commands perform C indentation and when.

If `c-auto-newline` is non-`nil`, newlines are inserted both before and after braces that you insert and after colons and semicolons. Correct C indentation is done on all the lines that are made this way.

If `c-tab-always-indent` is non-`nil`, the ⟨TAB⟩ command in C mode does indentation only if point is at the left margin or within the line's indentation. If there is non-whitespace to the left of point, ⟨TAB⟩ just inserts a tab character in the buffer. Normally, this variable is `nil`, and ⟨TAB⟩ always reindents the current line.

C does not have anything analogous to particular function names for which special forms of indentation are desirable. However, it has a different need for customization facilities: many different styles of C indentation are in common use.

There are six variables you can set to control the style that Emacs C mode will use.

`c-indent-level`
: Indentation of C statements within surrounding block. The surrounding block's indentation is the indentation of the line on which the open-brace appears.

`c-continued-statement-offset`
: Extra indentation given to a substatement, such as the then-clause of an `if` or body of a `while`.

`c-brace-offset`
: Extra indentation for lines that start 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 a line that is a label, 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. Another popular style prefers 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 non-blank character on the line. By setting the 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 line up exactly under the open brace. An exception are braces in column zero, like those surrounding a function's body. The statements inside those braces are not placed 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 inside other statements, like the then-clauses of `if` statements and the bodies of `while` statements. The `c-continued-statement-offset` parameter determines the difference in indentation between the two lines in:

```
if (x == y)
  do_it ();
```

The default value for `c-continued-statement-offset` 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 you set `c-brace-offset` to 4, the first example becomes:

```
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 and results in code like this:

```
char *
index (string, char)
     char *string;
     int char;
```

`c-label-offset` is the extra indentation given to a line that contains a label, a case statement, or a `default:` statement. Its standard value is −2 and results 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 default values.

Using the indentation style produced by the default settings of the variables just discussed and putting open braces on separate lines produces clear and readable files. For an example, look at any of the C source files of XEmacs.

22.5 Automatic Display of Matching Parentheses

The Emacs parenthesis-matching feature shows you automatically how parentheses match in the text. Whenever a self-inserting character that is a closing delimiter is typed, the cursor moves momentarily to the location of the matching opening delimiter, provided that is visible on the screen. If it is not on the screen, some text starting with that opening delimiter is displayed in the echo area. Either way, you see the grouping you are closing off.

In Lisp, automatic matching applies only to parentheses. In C, it also applies to braces and brackets. Emacs knows which characters to regard as matching delimiters based on the syntax table set by the major mode. See Section 29.5 [Syntax], page 274.

If the opening delimiter and closing delimiter are mismatched—as in '`[x)`'—the echo area displays a warning message. The correct matches are specified in the syntax table.

Two variables control parenthesis matching displays. `blink-matching-paren` turns the feature on or off. The default is `t` (match display is on); `nil` turns it off. `blink-matching-paren-distance` specifies how many characters back Emacs searches to find a matching opening delimiter. If the match is not found in the specified region, scanning stops, and nothing is displayed. This prevents wasting lots of time scanning when there is no match. The default is 4000.

22.6 Manipulating Comments

The comment commands insert, kill and align comments.

M-; Insert or align comment (`indent-for-comment`).

C-x ; Set comment column (`set-comment-column`).

C-u - C-x ;
Kill comment on current line (`kill-comment`).

M-⟨LFD⟩ Like ⟨RET⟩ followed by inserting and aligning a comment (`indent-new-comment-line`).

The command that creates a comment is *Meta-;* (`indent-for-comment`). If there is no comment already on the line, a new comment is created and aligned at a specific column called the *comment column*. Emacs creates the comment by inserting the string at the value of `comment-start`; see below. Point is left after that string. If the text of the line extends past the comment column, 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 string is inserted after point, to keep the syntax valid.

You can also use *Meta-;* to align an existing comment. If a line already contains the string that starts comments, *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 happens. To reinsert the comment on another line, move to the end of that line, type first *C-y*, and then *M-;* to realign the comment. 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 to call `kill-comment` when called with a negative argument. However, `kill-comment` is a valid command which you could bind directly to a key if you wanted to.

22.6.1 Multiple Lines of Comments

If you are typing a comment and want to continue it on another line, use the command *Meta-*⟨LFD⟩ (`indent-new-comment-line`), which terminates the comment you are typing, creates a new blank line afterward, and begins a new comment indented under the old one. If Auto Fill mode is on and you go past the fill column while typing, the comment is continued in just this fashion. If point is not at the end of the line when you type *M-*⟨LFD⟩, the text on the rest of the line becomes part of the new comment line.

22.6.2 Options Controlling Comments

The comment column is stored in the variable `comment-column`. You can explicitly set it to a number. 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 calls *Meta-;* to align the current line's comment under the previous one. Note that *C-u - C-x ;* runs the function `kill-comment` as described above.

`comment-column` is a per-buffer variable; altering the variable affects only the current buffer. You can also change the default value. See Section 29.2.4 [Locals], page 264. 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 89.)

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 and will follow the text you will insert into the comment. In C mode, `comment-start` has the value `"/* "` and `comment-end` has the value `" */"`.

`comment-multi-line` controls how *M-*⟨LFD⟩ (`indent-new-comment-line`) behaves when used inside a comment. If `comment-multi-line` is `nil`, as it normally is, then *M-*⟨LFD⟩ terminates the comment on the starting line and starts a new comment on the new following line. If `comment-multi-line` is not `nil`, then *M-*⟨LFD⟩ sets up the new following line 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 are legal, the value you choose for this variable is a matter of taste.

The variable `comment-indent-hook` should contain a function that is called to compute the indentation for a newly inserted comment or for aligning an existing comment. Major modes set this variable differently. 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. The function should return the column in which the comment ought to start. For example, in Lisp mode, the indent hook function bases its decision on the number of semicolons that begin an existing comment and on the code in the preceding lines.

22.7 Editing Without Unbalanced Parentheses

M-(Put parentheses around next sexp(s) (`insert-parentheses`).

M-) Move past next close parenthesis and re-indent (`move-over-close-and-reindent`).

The commands *M-(* (`insert-parentheses`) and *M-)* (`move-over-close-and-reindent`) are designed to facilitate a style of editing which keeps parentheses balanced at all times. *M-(* inserts a pair of parentheses, either together as in '`()`', or, if given an argument, around the next several sexps, and leaves point after the open parenthesis. Instead of typing *(F O O)*, you can type *M-(F O O*, which has the same effect except for leaving the cursor before the close parenthesis. You can then type *M-)*, which moves past the close parenthesis, deletes any indentation preceding it (in this example there is none), and indents with ⟨LFD⟩ after it.

22.8 Completion for Lisp Symbols

Completion usually happens in the minibuffer. An exception is completion for Lisp symbol names, which is available in all buffers.

The command *M-⟨TAB⟩* (`lisp-complete-symbol`) takes the partial Lisp symbol before point to be an abbreviation, and compares it against all non-trivial Lisp symbols currently known to Emacs. Any additional characters that they all have in common are inserted at point. Non-trivial symbols are those that have function definitions, values, or properties.

If there is an open-parenthesis immediately before the beginning of the partial symbol, only symbols with function definitions are considered as completions.

If the partial name in the buffer has more than one possible completion and they have no additional characters in common, a list of all possible completions is displayed in another window.

22.9 Documentation Commands

As you edit Lisp code to be run in Emacs, you can use the commands *C-h f* (`describe-function`) and *C-h v* (`describe-variable`) to print documentation of functions and variables 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.

The *M-x manual-entry* command gives you access to documentation on Unix commands, system calls, and libraries. The command reads a topic as an argument, and displays the Unix manual page for that topic. `manual-entry` always searches all 8 sections of the manual and concatenates all the entries it finds. For example, the topic '`termcap`' finds the description of the termcap library from section 3, followed by the description of the termcap data base from section 5.

22.10 Change Logs

The Emacs command *M-x add-change-log-entry* helps you keep a record of when and why you have changed a program. It assumes that you have a file in which you write a chronological sequence of entries describing individual changes. The default is to store the change entries in a file called '`ChangeLog`' in the same directory as the file you are editing. The same '`ChangeLog`' file therefore records changes for all the files in a directory.

A change log entry starts with a header line that contains your name and the current date. Except for these header lines, every line in the change log starts with a tab. One entry can describe several changes; each change starts with a line starting with a tab and a star. *M-x add-change-log-entry* visits the change log file and creates a new entry unless the most recent entry is for today's date and your name. In either case, it adds a new line to start the description of another change just after the header line of the entry. When *M-x add-change-log-entry* is finished, all is prepared for you to edit in the description of what you changed and how. You must then save the change log file yourself.

The change log file is always visited in Indented Text mode, which means that ⟨LFD⟩ and auto-filling indent each new line like the previous line. This is convenient for entering the contents of an entry, which must be indented. See Section 21.1 [Text Mode], page 149.

Here is an example of the formatting conventions used in the change log for Emacs:

```
Wed Jun 26 19:29:32 1985  Richard M. Stallman  (rms at mit-prep)

        * xdisp.c (try_window_id):
        If C-k is done at end of next-to-last line,
        this fn updates window_end_vpos and cannot leave
        window_end_pos nonnegative (it is zero, in fact).
        If display is preempted before lines are output,
        this is inconsistent.  Fix by setting
        blank_end_of_window to nonzero.

Tue Jun 25 05:25:33 1985  Richard M. Stallman  (rms at mit-prep)

        * cmds.c (Fnewline):
        Call the auto fill hook if appropriate.

        * xdisp.c (try_window_id):
        If point is found by compute_motion after xp, record that
        permanently.  If display_text_line sets point position wrong
        (case where line is killed, point is at eob and that line is
        not displayed), set it again in final compute_motion.
```

22.11 Tags Tables

A *tags 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 finds the definition of a function by looking up which of the files it is in.

Tags tables are stored in files called *tags table files*. The conventional name for a tags table file is '`TAGS`'.

Each entry in the tags 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 tags 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*.

22.11.1 Source File Tag Syntax

Here is how tag syntax is defined for the most popular languages:

- In C code, any C function or typedef is a tag, and so are definitions of `struct`, `union` and `enum`. `#define` macro definitions and `enum` constants are also tags, unless you specify '`--no-defines`' when making the tags table. Similarly, global variables are tags, unless you specify '`--no-globals`'. Use of '`--no-globals`' and '`--no-defines`' can make the tags table file much smaller.
- In C++ code, in addition to all the tag constructs of C code, member functions are also recognized, and optionally member variables if you use the '`--members`' option. Tags for variables and functions in classes are named '*class*`::`*variable*' and '*class*`::`*function*'.
- In Java code, tags include all the constructs recognized in C++, plus the `extends` and `implements` constructs. Tags for variables and functions in classes are named '*class*`.`*variable*' and '*class*`.`*function*'.
- In LaTeX text, the argument of any of the commands `\chapter`, `\section`, `\subsection`, `\subsubsection`, `\eqno`, `\label`, `\ref`, `\cite`, `\bibitem`, `\part`, `\appendix`, `\entry`, or `\index`, is a tag.

  Other commands can make tags as well, if you specify them in the environment variable `TEXTAGS` before invoking `etags`. The value of this environment variable should be a colon-separated list of commands names. For example,

  ```
  TEXTAGS="def:newcommand:newenvironment"
  export TEXTAGS
  ```

  specifies (using Bourne shell syntax) that the commands '`\def`', '`\newcommand`' and '`\newenvironment`' also define tags.
- 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.

Several other languages are also supported:

- In assembler code, labels appearing at the beginning of a line, followed by a colon, are tags.
- In Bison or Yacc 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 Cobol code, paragraphs names are the tags, i.e. any word starting in column 8 and followed by a full stop.
- In Erlang code, the tags are the functions, records, and macros defined in the file.
- In Fortran code, functions and subroutines are tags.
- In Objective C code, tags include Objective C definitions for classes, class categories, methods and protocols.
- In Pascal code, the tags are the functions and procedures defined in the file.
- In Perl code, the tags are the procedures defined by the `sub` keyword.
- In Postscript code, the tags are the functions.
- In Prolog code, a tag name appears at the left margin.

You can also generate tags based on regexp matching (see Section 22.11.2 [Create Tags Table], page 177) to handle other formats and languages.

22.11.2 Creating Tags Tables

The `etags` program is used to create a tags 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 tags table named '`TAGS`' in the current working directory. `etags` recognizes the language used in an input file based on its file name and contents. You can specify the language with the '`--language=`*name*' option, described below.

If the tags table data become outdated due to changes in the files described in the table, the way to update the tags table is the same way it was made in the first place. It is not necessary to do this often.

If the tags 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 tags 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 tags 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 tags table after each edit, or even every day.

One tags table can effectively include another. Specify the included tags file name with the '`--include=`*file*' option when creating the file that is to include it. The latter file then acts as if it contained all the files specified in the included file, as well as the files it directly contains.

If you specify the source files with relative file names when you run `etags`, the tags file will contain file names relative to the directory where the tags file was initially written. This way, you can move an entire directory tree containing both the tags file and the source files, and the tags file will still refer correctly to the source files.

If you specify absolute file names as arguments to `etags`, then the tags file will contain absolute file names. This way, the tags file will still refer to the same files even if you move it, as long as the source files remain in the same place. Absolute file names start with '`/`', or with '*device*`:/`' on MS-DOS and Windows.

When you want to make a tags table from a great number of files, you may have problems listing them on the command line, because some systems have a limit on its length. The simplest way to circumvent this limit is to tell `etags` to read the file names from its standard input, by typing a dash in place of the file names, like this:

```
find . -name "*.[chCH]" -print | etags -
```

Use the option '`--language=`*name*' to specify the language explicitly. You can intermix these options with file names; each one applies to the file names that follow it. Specify '`--language=auto`' to tell `etags` to resume guessing the language from the file names and file contents. Specify '`--language=none`' to turn off language-specific processing entirely; then `etags` recognizes tags by regexp matching alone. '`etags --help`' prints the list of the languages `etags` knows, and the file name rules for guessing the language.

The '`--regex`' option provides a general way of recognizing tags based on regexp matching. You can freely intermix it with file names. Each '`--regex`' option adds to the preceding ones, and applies only to the following files. The syntax is:

```
--regex=/tagregexp[/nameregexp]/
```

where *tagregexp* is used to match the lines to tag. It is always anchored, that is, it behaves as if preceded by '`^`'. If you want to account for indentation, just match any initial number of blanks by beginning your regular expression with '`[ \t]*`'. In the regular expressions, '`\`' quotes the next character, and '`\t`' stands for the tab character. Note that `etags` does not handle the other C escape sequences for special characters.

The syntax of regular expressions in `etags` is the same as in Emacs, augmented with the *interval operator*, which works as in `grep` and `ed`. The syntax of an interval operator is '`\{`*m*`,`*n*`\}`', and its meaning is to match the preceding expression at least *m* times and up to *n* times.

You should not match more characters with *tagregexp* than that needed to recognize what you want to tag. If the match is such that more characters than needed are unavoidably matched by *tagregexp*, you may find useful to add a *nameregexp*, in order to narrow the tag scope. You can find some examples below.

The '`-R`' option deletes all the regexps defined with '`--regex`' options. It applies to the file names following it, as you can see from the following example:

```
etags --regex=/reg1/ voo.doo --regex=/reg2/ \
    bar.ber -R --lang=lisp los.er
```

Here `etags` chooses the parsing language for '`voo.doo`' and '`bar.ber`' according to their contents. `etags` also uses *reg1* to recognize additional tags in '`voo.doo`', and both *reg1* and *reg2* to recognize additional tags in '`bar.ber`'. `etags` uses the Lisp tags rules, and no regexp matching, to recognize tags in '`los.er`'.

Here are some more examples. The regexps are quoted to protect them from shell interpretation.

Tag the `DEFVAR` macros in the emacs source files:

```
--regex='/[ \t]*DEFVAR_[A-Z_ \t(]+"\([^"]+\)"/'
```

Tag VHDL files (this example is a single long line, broken here for formatting reasons):

```
--language=none
--regex='/[ \t]*\(ARCHITECTURE\|CONFIGURATION\) +[^ ]* +OF/'
--regex='/[ \t]*\(ATTRIBUTE\|ENTITY\|FUNCTION\|PACKAGE\
\( BODY\)?\|PROCEDURE\|PROCESS\|TYPE\)[ \t]+\([^ \t(]+\)/\3/'
```

Tag TCL files (this last example shows the usage of a *nameregexp*):

```
--lang=none --regex='/proc[ \t]+\([^ \t]+\)/\1/'
```

For a list of the other available `etags` options, execute `etags --help`.

22.11.3 Selecting a Tags Table

At any time Emacs has one *selected* tags table, and all the commands for working with tags tables use the selected one. To select a tags table, use the variable `tag-table-alist`.

The value of `tag-table-alist` is a list that determines which `TAGS` files should be active for a given buffer. This is not really an association list, in that all elements are checked. The car of each element of this list is a pattern against which the buffers file name is compared; if it matches, then the cdr of the list should be the name of the tags table to use. If more than one element of this list matches the buffers file name, all of the associated tags tables are used. Earlier ones are searched first.

If the car of elements of this list are strings, they are treated as regular-expressions against which the file is compared (like the `auto-mode-alist`). If they are not strings, they are evaluated. If they evaluate to non-`nil`, the current buffer is considered to match.

If the cdr of the elements of this list are strings, they are assumed to name a tags file. If they name a directory, the string '`tags`' is appended to them to get the file name. If they are not strings, they are evaluated and must return an appropriate string.

For example:

```
  (setq tag-table-alist
'(("/usr/src/public/perl/" . "/usr/src/public/perl/perl-3.0/")
```

```
("\\.el$" . "/usr/local/emacs/src/")
("/jbw/gnu/" . "/usr15/degree/stud/jbw/gnu/")
("" . "/usr/local/emacs/src/")
))
```

The example defines the tags table alist in the following way:

- Anything in the directory '`/usr/src/public/perl/`' should use the '`TAGS`' file '`/usr/src/public/perl/perl-3.0/TAGS`'.
- Files ending in '`.el`' should use the '`TAGS`' file '`/usr/local/emacs/src/TAGS`'.
- Anything in or below the directory '`/jbw/gnu/`' should use the '`TAGS`' file '`/usr15/degree/stud/jbw/gnu/TAGS`'.

If you had a file called '`/usr/jbw/foo.el`', it would use both '`TAGS`' files, '`/usr/local/emacs/src/TAGS`' and '`/usr15/degree/stud/jbw/gnu/TAGS`' (in that order), because it matches both patterns.

If the buffer-local variable `buffer-tag-table` is set, it names a tags table that is searched before all others when `find-tag` is executed from this buffer.

If there is a file called '`TAGS`' in the same directory as the file in question, then that tags file will always be used as well (after the `buffer-tag-table` but before the tables specified by this list).

If the variable `tags-file-name` is set, the '`TAGS`' file it names will apply to all buffers (for backwards compatibility.) It is searched first.

If the value of the variable `tags-always-build-completion-table` is `t`, the tags file will always be added to the completion table without asking first, regardless of the size of the tags file.

The function *`M-x visit-tags-table`*, is largely made obsolete by the variable `tag-table-alist`, tells tags commands to use the tags table file *file* first. The *file* should be the name of a file created with the `etags` program. A directory name is also acceptable; it means the file '`TAGS`' in that directory. The function only stores the file name you provide in the variable `tags-file-name`. Emacs does not actually read in the tags table contents until you try to use them. You can set the variable explicitly instead of using `visit-tags-table`. The value of the variable `tags-file-name` is the name of the tags table used by all buffers. This is for backward compatibility, and is largely supplanted by the variable `tag-table-alist`.

22.11.4 Finding a Tag

The most important thing that a tags table enables you to do is to find the definition of a specific tag.

`M-.` *tag &optional other-window*
: Find first definition of *tag* (`find-tag`).

`C-u M-.`
: Find next alternate definition of last tag specified.

`C-x 4 .` *tag*
: Find first definition of *tag*, but display it in another window (`find-tag-other-window`).

M-. (`find-tag`) is the command to find the definition of a specified tag. It searches through the tags table for that tag, as a string, then uses the tags table information to determine the file in which the definition is used and the approximate character position of the definition in the file. Then `find-tag` visits the file, moves point to the approximate character position, and starts searching ever-increasing distances away for the text that should appear at the beginning of the definition.

If an empty argument is given (by typing ⟨RET⟩), the sexp in the buffer before or around point is used as the name of the tag to find. See Section 22.2 [Lists], page 164, for information on sexps.

The argument to `find-tag` need not be the whole tag name; it can be a substring of a tag name. However, there can be many tag names containing the substring you specify. Since `find-tag` works by searching the text of the tags table, it finds the first tag in the table that the specified substring appears in. To find other tags that match the 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 your keyboard has a real ⟨META⟩ key, *M-0 M-.* is an easier alternative to *C-u M-.*.

If the optional second argument *other-window* is non-`nil`, it uses another window to display the tag. Multiple active tags tables and completion are supported.

Variables of note include the following:

tag-table-alist
: Controls which tables apply to which buffers.

tags-file-name
: Stores a default tags table.

tags-build-completion-table
: Controls completion behavior.

buffer-tag-table
: Specifies a buffer-local table.

make-tags-files-invisible
: Sets whether tags tables should be very hidden.

tag-mark-stack-max
: Specifies how many tags-based hops to remember.

Like most commands that can switch buffers, `find-tag` has another similar command that displays the new buffer in another window. *C-x 4 .* invokes the function `find-tag-other-window`. (This key sequence ends with a period.)

Emacs comes with a tags table file '`TAGS`' (in the directory containing Lisp libraries) 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 look at the source of any Emacs function.

22.11.5 Searching and Replacing with Tags Tables

The commands in this section visit and search all the files listed in the selected tags table, one by one. For these commands, the tags table serves only to specify a sequence of files to search. A related command is `M-x grep` (see Section 23.1 [Compilation], page 189).

`M-x tags-search` ⟨RET⟩ *regexp* ⟨RET⟩
: Search for *regexp* through the files in the selected tags table.

`M-x tags-query-replace` ⟨RET⟩ *regexp* ⟨RET⟩ *replacement* ⟨RET⟩
: Perform a `query-replace-regexp` on each file in the selected tags 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 tags 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 tags table.

`M-x tags-query-replace` performs a single `query-replace-regexp` through all the files in the tags 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 92, for more information on query replace.

It is possible to get through all the files in the tags table with a single invocation of `M-x tags-query-replace`. But often it is useful to exit temporarily, which you can do with any input event that has no special query replace meaning. You can resume the query replace subsequently by typing `M-,`; this command resumes the last tags search or replace command that you did.

The commands in this section carry out much broader searches than the `find-tag` family. The `find-tag` 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 much like running a compilation; finding the source locations of the `grep` matches works like finding the compilation errors. See Section 23.1 [Compilation], page 189.

If you wish to process all the files in a selected tags table, but `M-x tags-search` and `M-x tags-query-replace` are not giving you the desired result, you can use `M-x next-file`.

C-u M-x `next-file`
With a numeric argument, regardless of its value, visit the first file in the tags table and prepare to advance sequentially by files.

M-x `next-file`
Visit the next file in the selected tags table.

22.11.6 Tags 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 tags 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 tags table; it is read as a string rather than 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 tags table contains that directory.

M-x `tags-apropos` is like `apropos` for tags. It reads a regexp, then finds all the tags in the selected tags table whose entries match that regexp, and displays the tag names found.

22.12 Fortran Mode

Fortran mode provides special motion commands for Fortran statements and subprograms, and indentation commands that understand Fortran conventions of nesting, line numbers, and continuation statements.

Special commands for comments are provided because Fortran comments are unlike those of other languages.

Built-in abbrevs optionally save typing when you insert Fortran keywords.

Use *M-x* `fortran-mode` to switch to this major mode. Doing so calls the value of `fortran-mode-hook` as a function of no arguments if that variable has a non-`nil` value.

Fortran mode was contributed by Michael Prange.

22.12.1 Motion Commands

Fortran mode provides special commands to move by subprograms (functions and subroutines) and by statements. There is also a command to put the region around one subprogram, which is 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`).

22.12.2 Fortran Indentation

Special commands and features are available for indenting Fortran code. They make sure various syntactic entities (line numbers, comment line indicators, and continuation line flags) appear in the columns that are required for standard Fortran.

22.12.2.1 Fortran Indentation Commands

⟨TAB⟩ Indent the current line (`fortran-indent-line`).

M-⟨LFD⟩ Break the current line and set up a continuation line.

C-M-q Indent all the lines of the subprogram point is in (`fortran-indent-subprogram`).

⟨TAB⟩ is redefined by Fortran mode to reindent the current line for Fortran (`fortran-indent-line`). Line numbers and continuation markers are indented to their required columns, and the body of the statement is independently indented, based on its nesting in the program.

The key *C-M-q* is redefined as `fortran-indent-subprogram`, a command that reindents all the lines of the Fortran subprogram (function or subroutine) containing point.

The key *M-*⟨LFD⟩ is redefined as `fortran-split-line`, a command to split a line in the appropriate fashion for Fortran. In a non-comment line, the second half becomes a continuation line and is indented accordingly. In a comment line, both halves become separate comment lines.

22.12.2.2 Line Numbers and Continuation

If a number is the first non-whitespace in the line, it is assumed to be a line number and is moved to columns 0 through 4. (Columns are always counted from 0 in XEmacs.) If the text on the line starts with the conventional Fortran continuation marker '`$`', it is moved to column 5. If the text begins with any non whitespace character in column 5, it is assumed to be an unconventional continuation marker and remains in column 5.

Line numbers of four digits or less are normally indented one space. This amount is controlled by the variable `fortran-line-number-indent`, which is the maximum indentation a line number can have. Line numbers are indented to right-justify them to end in column 4 unless that would require more than the 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.

22.12.2.3 Syntactic Conventions

Fortran mode assumes that you follow certain conventions that simplify the task of understanding a Fortran program well enough to indent it properly:

- Two nested '`do`' loops never share a '`continue`' statement.
- The same character appears in column 5 of all continuation lines. It is the value of the variable `fortran-continuation-char`. By default, this character is '`$`'.

If you fail to follow these conventions, the indentation commands may indent some lines unaesthetically. However, a correct Fortran program will retain its meaning when reindented even if the conventions are not followed.

22.12.2.4 Variables for Fortran Indentation

Several additional variables control how Fortran indentation works.

`fortran-do-indent`
: Extra indentation within each level of '`do`' statement (the default is 3).

`fortran-if-indent`
: Extra indentation within each level of '`if`' statement (the default is 3).

`fortran-continuation-indent`
: Extra indentation for bodies of continuation lines (the default is 5).

`fortran-check-all-num-for-matching-do`
: If this is `nil`, indentation assumes that each '`do`' statement ends on a '`continue`' statement. Therefore, when computing indentation for a statement other than '`continue`', it can save time by not checking for a '`do`' statement ending there. If this is non-`nil`, indenting any numbered statement must check for a '`do`' that ends there. The default is `nil`.

`fortran-minimum-statement-indent`
: Minimum indentation for Fortran statements. For standard Fortran, this is 6. Statement bodies are always indented at least this much.

22.12.3 Comments

The usual Emacs comment commands assume that a comment can follow a line of code. In Fortran, the standard comment syntax requires an entire line to be just a comment. Therefore, Fortran mode replaces the standard Emacs comment commands and defines some new variables.

Fortran mode can also handle a non-standard 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 specified to do so in advance by setting the variable `comment-start` to '`"!"`' (see Section 29.2 [Variables], page 257).

M-; Align comment or insert new comment (`fortran-comment-indent`).

C-x ; Applies to nonstandard '`!`' comments only.

C-c ; Turn all lines of the region into comments, or (with arg) turn them back into real code (`fortran-comment-region`).

M-; in Fortran mode is redefined as the command `fortran-comment-indent`. Like the usual *M-;* command, it recognizes an existing comment and aligns its text appropriately. If there is no existing comment, a comment is inserted and aligned.

Inserting and aligning comments is not the same in Fortran mode as in other modes. When a new comment must be inserted, a full-line comment is inserted if the current line is blank. On a non-blank line, a non-standard '`!`' comment is inserted if you previously specified you wanted to use them. Otherwise a full-line comment is inserted on a new line before the current line.

Non-standard '`!`' comments are aligned like comments in other languages, but full-line comments are aligned differently. In a standard full-line comment, the comment delimiter itself must always appear in column zero. What can be aligned is the text within the comment. You can choose from three styles of alignment by setting the variable `fortran-comment-indent-style` to one of these values:

`fixed` The text is aligned at a fixed column, which is the value of `fortran-comment-line-column`. This is the default.

`relative` The text is aligned as if it were a line of code, but with an additional `fortran-comment-line-column` columns of indentation.

`nil` Text in full-line columns is not moved automatically.

You can also specify the character to be used to indent within full-line comments by setting the variable `fortran-comment-indent-char` to the character you want to use.

Fortran mode introduces two variables `comment-line-start` and `comment-line-start-skip`, which do for full-line comments what `comment-start` and `comment-start-skip` do 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. It can therefore be used if you use '`!`' comments, but is useless in Fortran mode otherwise.

The command *C-c ;* (`fortran-comment-region`) turns all the lines of the region into comments by inserting the string '`C$$$`' at the front of each one. With a numeric arg, the region is turned back into live code by deleting '`C$$$`' from the front of each line. You can control the string used for the comments by setting the variable `fortran-comment-region`. Note that here we have an example of a command and a variable with the same name; the two uses of the name never conflict because in Lisp and in Emacs it is always clear from the context which one is referred to.

22.12.4 Columns

C-c C-r Displays a "column ruler" momentarily above the current line (`fortran-column-ruler`).

C-c C-w Splits the current window horizontally so that it is 72 columns wide. This may help you avoid going over that limit (`fortran-window-create`).

The command *C-c C-r* (`fortran-column-ruler`) shows a column ruler above the current line. The comment ruler consists of 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 XEmacs. As a result, the numbers may not be those you are familiar with; but the actual positions in the line are standard Fortran.

The text used to display the column ruler is the value of the variable `fortran-comment-ruler`. By changing this variable, you can change the display.

For even more help, use *C-c C-w* (`fortran-window-create`), a command which splits the current window horizontally, resulting in a window 72 columns wide. When you edit in this window, you can immediately see when a line gets too wide to be correct Fortran.

22.12.5 Fortran Keyword Abbrevs

Fortran mode provides many built-in abbrevs for common keywords and declarations. These are the same sort of abbrevs that you can define yourself. To use them, you must turn on Abbrev mode. see Chapter 24 [Abbrevs], page 203.

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`' changes automatically to '`continue`', provided Abbrev mode is enabled.

Type '`;?`' or '`;C-h`' to display a list of all built-in Fortran abbrevs and what they stand for.

22.13 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 non-interactive languages like C and Fortran as inferior processes, feeding the error log into an Emacs buffer. It can also parse the error messages and visit the files in which errors are found, moving point to the line where the error occurred.

M-x compile
: Run a compiler asynchronously under Emacs, with error messages to '`*compilation*`' buffer.

M-x grep
: Run `grep` asynchronously under Emacs, with matching lines listed in the buffer named '`*compilation*`'.

M-x kill-compilation
: Kill the process made by the `M-x compile` command.

M-x kill-grep
: Kill the running compilation or `grep` subprocess.

C-x `
: Visit the next compiler error message or `grep` match.

To run `make` or another compiler, type *M-x compile*. This command reads a shell command line using the minibuffer, then executes the specified command line in an inferior shell with output going to the buffer named '`*compilation*`'. By default, the current buffer's default directory is used as the working directory for the execution of the command; therefore, the makefile comes from this directory.

When the shell command line is read, the minibuffer appears containing a default command line (the command you used the last time you typed *M-x compile*). If you type just ⟨RET⟩, the same command line is used again. The first *M-x compile* provides `make -k` as the default. The default is taken from the variable `compile-command`; if the appropriate compilation command for a file is something other than `make -k`, it can be useful to have the file specify a local value for `compile-command` (see Section 29.2.5 [File Variables], page 266).

When you start a compilation, the buffer '`*compilation*`' is displayed in another window but not selected. Its mode line displays the word '`run`' or '`exit`' in the parentheses to tell you whether compilation is finished. You do not have to keep this buffer visible; compilation continues in any case.

To kill the compilation process, type *M-x-kill-compilation*. 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 occur at any time. Starting a new compilation prompts for confirmation before actually killing a compilation that is running.

To parse the compiler error messages, type *C-x `* (`next-error`). The character following *C-x* is the grave accent, not the single quote. The command displays the buffer '`*compilation*`' in one window and the buffer in which the next error occurred in another window. Point in that buffer is moved to the line where the error was found. The corresponding error message is scrolled to the top of the window in which '`*compilation*`' is displayed.

The first time you use *C-x `* after the start of a compilation, it parses all the error messages, visits all the files that have error messages, and creates markers pointing at the lines the error messages refer to. It then moves to the first error message location. Subsequent uses of *C-x `* advance down the data set up by the first use. When the preparsed error messages are exhausted, the next *C-x `* checks for any more error messages that have come in; this is useful if you start editing compiler errors while compilation is still going on. If no additional 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 again, then displays 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 to `grep`: a `grep`-style regexp (usually in single quotes to quote the shell's special characters) followed by filenames, which may use wildcard characters. The output from `grep` goes in the '`*compilation*`' buffer. You can use *C-x `* to find the lines that match as if they were compilation errors.

Note: a shell is used to run the compile command, but the shell is not run in interactive mode. In particular, this means 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 initialization file ('`.cshrc`' or '`.shrc`' or ...) by setting the prompt unconditionally. The shell initialization file should set the prompt only if there already is a prompt. Here's how to do it in `csh`:

```
if ($?prompt) set prompt = ...
```

23.2 Major Modes for Lisp

Emacs has four different major modes for Lisp. They are the same in terms of editing commands, but differ in the commands for executing Lisp expressions.

Emacs-Lisp mode
: The mode for editing source files of programs to run in Emacs Lisp. This mode defines *C-M-x* to evaluate the current defun. See Section 23.3 [Lisp Libraries], page 191.

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 196.

Lisp mode
: The mode for editing source files of programs that run in other dialects of Lisp 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 196.

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 28.2.3 [Shell Mode], page 250).

Scheme mode
: Like Lisp mode but for Scheme programs.

Inferior Scheme mode
: The mode for an interactive session with an inferior Scheme process.

23.3 Libraries of Lisp Code for Emacs

Lisp code for Emacs editing commands is stored in files whose names conventionally end in '`.el`'. This ending tells Emacs to edit them in Emacs-Lisp mode (see Section 23.2 [Lisp Modes], page 190).

23.3.1 Loading Libraries

`M-x load-file` *file*
: Load the file *file* of Lisp code.

`M-x load-library` *library*
: Load the library named *library*.

`M-x locate-library` *library* *`&optional`* *nosuffix*
: Show the full path name of Emacs library *library*.

To execute a file of Emacs Lisp, use *`M-x load-file`*. This command reads the file name you provide in the minibuffer, then executes the contents of that file as Lisp code. It is not necessary to visit the file first; in fact, this command reads the file as found on disk, not the text in an Emacs buffer.

Once a file of Lisp code is installed in the Emacs Lisp library directories, users can load it using *`M-x load-library`*. Programs can load it by calling `load-library`, or with `load`, a more primitive function that is similar but accepts some additional arguments.

`M-x load-library` differs from *`M-x load-file`* in that it searches a sequence of directories and tries three file names in each directory. The three names are: first, the specified name with '`.elc`' appended; second, the name with '`.el`' appended; third, the specified name alone. A '`.elc`' file would be the result of compiling the Lisp file into byte code; if possible, it is loaded in preference to the Lisp file itself because the compiled file loads and runs faster.

Because the argument to `load-library` is usually not in itself a valid file name, file name completion is not available. In fact, 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 elements of this list may not begin with "'~'", so you must call `expand-file-name` on them before adding them to the list. 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 start wishing that `nil` were in the list, you should probably use *M-x load-file* for this case.

The variable is initialized by the **EMACSLOADPATH** environment variable. If no value is specified, the variable takes the default value specified in the file '`paths.h`' when Emacs was built. If a path isn't specified in '`paths.h`', a default value is obtained from the file system, near the directory in which the Emacs executable resides.

Like *M-x load-library*, *M-x locate-library* searches the directories in `load-path` to find the file that *M-x load-library* would load. If the optional second argument *nosuffix* is non-`nil`, the suffixes '`.elc`' or '`.el`' are not added to the specified name *library* (like calling `load` instead of `load-library`).

You often 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, the entire file is loaded a second time if you use one of the autoloadable commands again. This prevents problems when the command is no longer autoloading but is working incorrectly because the file was only partially loaded. Function definitions are undone only for autoloading; explicit calls to `load` do not undo anything if loading is not completed.

The variable `after-load-alist` takes an alist of expressions to be evaluated when particular files are loaded. Each element has the form `(`*filename* `forms...)`. When `load` is run and the filename argument is *filename*, the forms in the corresponding element are executed at the end of loading.

filename must match exactly. Normally *filename* is the name of a library, with no directory specified, since that is how load is normally called. An error in `forms` does not undo the load, but it does prevent execution of the rest of the `forms`.

23.3.2 Compiling Libraries

Emacs Lisp code can be compiled into byte-code which loads faster, takes up less space when loaded, and executes faster.

M-x batch-byte-compile
: Run byte-compile-file on the files remaining on the command line.

M-x byte-compile-buffer &optional buffer
: Byte-compile and evaluate contents of *buffer* (default is current buffer).

M-x byte-compile-file
: Compile a file of Lisp code named *filename* into a file of byte code.

M-x byte-compile-and-load-file filename
: Compile a file of Lisp code named *filename* into a file of byte code and load it.

M-x byte-recompile-directory directory
: Recompile every '`.el`' file in *directory* that needs recompilation.

M-x disassemble
: Print disassembled code for *object* on (optional) *stream*.

M-x make-obsolete function new
: Make the byte-compiler warn that *function* is obsolete and *new* should be used instead.

byte-compile-file creates a byte-code compiled file from an Emacs-Lisp source file. The default argument for this function is the file visited in the current buffer. The function reads the specified file, compiles it into byte code, and writes an output file whose name is made by appending 'c' to the input file name. Thus, the file 'rmail.el' would be compiled into 'rmail.elc'. To compile a file of Lisp code named *filename* into a file of byte code and then load it, use byte-compile-and-load-file. To compile and evaluate Lisp code in a given buffer, use byte-compile-buffer.

To recompile all changed Lisp files in a directory, use *M-x byte-recompile-directory*. Specify just the directory name as an argument. Each '.el' file that has been byte-compiled before is byte-compiled again if it has changed since the previous compilation. A numeric argument to this command tells it to offer to compile each '.el' file that has not been compiled yet. You must answer *y* or *n* to each offer.

You can use the function batch-byte-compile to invoke Emacs non-interactively from the shell to do byte compilation. When you use this function, the files to be compiled are specified with command-line arguments. Use a shell command of the form:

```
emacs -batch -f batch-byte-compile files...
```

Directory names may also be given as arguments; in that case, byte-recompile-directory is invoked on each such directory. batch-byte-compile uses all remaining command-line arguments as file or directory names, then kills the Emacs process.

M-x disassemble explains the result of byte compilation. Its argument is a function name. It displays the byte-compiled code in a help window in symbolic form, one instruction per line. If the instruction refers to a variable or constant, that is shown, too.

23.3.3 Converting Mocklisp to Lisp

XEmacs can run Mocklisp files by converting them to Emacs Lisp first. To convert a Mocklisp file, visit it and then type *M-x convert-mocklisp-buffer*. Then save the resulting buffer of Lisp file in a file whose name ends in '.el' and use the new file as a Lisp library.

You cannot currently byte-compile converted Mocklisp code. The reason is that converted Mocklisp code uses some special Lisp features to deal with Mocklisp's incompatible ideas of how arguments are evaluated and which values signify "true" or "false".

23.4 Evaluating Emacs-Lisp Expressions

Lisp programs intended to be run in Emacs should be edited in Emacs-Lisp mode; this will happen automatically for file names ending in '.el'. By contrast, Lisp mode itself should be used for editing Lisp programs intended for other Lisp systems. Emacs-Lisp mode can be selected with the command *M-x emacs-lisp-mode*.

For testing of Lisp programs to run in Emacs, it is useful to be able to evaluate part of the program as it is found in the Emacs buffer. For example, if you change the text of a Lisp function definition and then evaluate the definition, Emacs installs the change for future calls to the function. Evaluation of Lisp expressions is also useful in any kind of editing task for invoking non-interactive functions (functions that are not commands).

M-⟨ESC⟩
: Read a Lisp expression in the minibuffer, evaluate it, and print the value in the minibuffer (`eval-expression`).

C-x C-e
: Evaluate the Lisp expression before point, and print the value in the minibuffer (`eval-last-sexp`).

C-M-x
: Evaluate the defun containing point or after point, and print the value in the minibuffer (`eval-defun`).

M-x eval-region
: Evaluate all the Lisp expressions in the region.

M-x eval-current-buffer
: Evaluate all the Lisp expressions in the buffer.

M-⟨ESC⟩ (`eval-expression`) is the most basic command for evaluating a Lisp expression interactively. It reads the expression using the minibuffer, so you can execute any expression on a buffer regardless of what the buffer contains. When evaluation is complete, the current buffer is once again the buffer that was current when *M-*⟨ESC⟩ was typed.

M-⟨ESC⟩ can easily confuse users, 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, you are no longer required to confirm. See Section 29.4.3 [Disabling], page 274.

In Emacs-Lisp mode, the key *C-M-x* is bound to the function `eval-defun`, which parses the defun containing point 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* are given a numeric argument, they print the value by inserting it 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 it 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.

23.5 The Emacs-Lisp Debugger

XEmacs contains a debugger for Lisp programs executing inside it. This debugger is normally not used; many commands frequently get Lisp errors when invoked in inappropriate contexts (such as `C-f` at the end of the buffer) and it would be unpleasant to enter a special debugging mode in this case. When you want to make Lisp errors invoke the debugger, you must set the variable `debug-on-error` to non-`nil`. Quitting with `C-g` is not considered an error, and `debug-on-error` has no effect on the handling of `C-g`. However, if you set `debug-on-quit` to be non-`nil`, `C-g` will invoke the debugger. This can be useful for debugging an infinite loop; type `C-g` once the loop has had time to reach its steady state. `debug-on-quit` has no effect on errors.

You can make Emacs enter the debugger when a specified function is called or at a particular place in Lisp code. Use `M-x debug-on-entry` with argument *fun-name* to have Emacs enter the debugger as soon as *fun-name* is called. Use `M-x cancel-debug-on-entry` to make the function stop entering the debugger when called. (Redefining the function also does this.) To enter the debugger from some other place in Lisp code, you must insert the expression `(debug)` there and install the changed code with `C-M-x`. See Section 23.4 [Lisp Eval], page 193.

When the debugger is entered, it displays the previously selected buffer in one window and a buffer named '`*Backtrace*`' in another window. The backtrace buffer contains one line for each level of Lisp function execution currently going on. At the beginning of the buffer is a message describing the reason that the debugger was invoked, for example, an error message if it was invoked due to an error.

The backtrace buffer is read-only and is in Backtrace mode, a special major mode in which letters are defined as debugger commands. The usual Emacs editing commands are available; you can switch windows to examine the buffer that was being edited at the time of the error, and you can switch buffers, visit files, and perform any other editing operations. However, the debugger is a recursive editing level (see Section 28.5 [Recursive Edit], page 253); it is a good idea to return to the backtrace buffer and explictly exit the debugger when you don't want to use it any more. Exiting the debugger kills the backtrace buffer.

The contents of the backtrace buffer show you the functions that are executing and the arguments that were given to them. It also allows you to specify a stack frame by moving point to the line describing that frame. The frame whose line point is on is considered the *current frame*. Some of the debugger commands operate on the current frame. Debugger commands are mainly used for stepping through code one expression at a time. Here is a list of them:

`c`
: Exit the debugger and continue execution. In most cases, execution of the program continues as if the debugger had never been entered (aside from the effect of any variables or data structures you may have changed while inside the debugger). This includes entry to the debugger due to function entry or exit, explicit invocation, and quitting or certain errors. Most errors cannot be continued; trying to continue an error usually causes the same error to occur again.

`d`
: Continue execution, but enter the debugger the next time a Lisp function is called. This allows you to step through the subexpressions of an expression, and see what the subexpressions do and what values they compute.

  When you enter the debugger this way, Emacs flags the stack frame for the function call from which you entered. The same function is then called when you exit the frame. To cancel this flag, use `u`.

b Set up to enter the debugger when the current frame is exited. Frames that invoke the debugger on exit are flagged with stars.

u Don't enter the debugger when the current frame is exited. This cancels a *b* command on a frame.

e Read a Lisp expression in the minibuffer, evaluate it, and print the value in the echo area. This is equivalent to the command *M-*⟨ESC⟩, except that *e* is not normally disabled like *M-*⟨ESC⟩.

q Terminate the program being debugged; return to top-level Emacs command execution.

If the debugger was entered due to a *C-g* but you really want to quit, not to debug, use the *q* command.

r Return a value from the debugger. The value is computed by reading an expression with the minibuffer and evaluating it.

The value returned by the debugger makes a difference when the debugger was invoked due to exit from a Lisp call frame (as requested with *b*); then the value specified in the *r* command is used as the value of that frame.

The debugger's return value also matters with many errors. For example, `wrong-type-argument` errors will use the debugger's return value instead of the invalid argument; `no-catch` errors will use the debugger value as a throw tag instead of the tag that was not found. If an error was signaled by calling the Lisp function `signal`, the debugger's return value is returned as the value of `signal`.

23.6 Lisp Interaction Buffers

The buffer '`*scratch*`', which is selected when Emacs starts up, is provided for evaluating Lisp expressions interactively inside Emacs. Both the expressions you evaluate and their output goes in the buffer.

The '`*scratch*`' buffer's major mode is Lisp Interaction mode, which is the same as Emacs-Lisp mode except for one command, ⟨LFD⟩. In Emacs-Lisp mode, ⟨LFD⟩ is an indentation command. In Lisp Interaction mode, ⟨LFD⟩ is bound to `eval-print-last-sexp`. This function reads the Lisp expression before point, evaluates it, and inserts the value in printed representation before point.

The way to use the '`*scratch*`' buffer is to insert Lisp expressions at the end, ending each one with ⟨LFD⟩ so that it will be evaluated. The result is a complete typescript of the expressions you have evaluated and their values.

The rationale for this feature is that Emacs must have a buffer when it starts up, but that buffer is not useful for editing files since a new buffer is made for every file that you visit. The Lisp interpreter typescript is the most useful thing I can think of for the initial buffer to do. *M-x lisp-interaction-mode* will put any buffer in Lisp Interaction mode.

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*`'. In other words, any "terminal output" from Lisp will go into the buffer, advancing point, and any "terminal input" for Lisp comes from text in the buffer. To give input to Lisp, go to the end of the buffer and type the input, terminated by ⟨RET⟩. The '`*lisp*`' buffer is in Inferior Lisp mode, which has all the special characteristics of Lisp mode and Shell mode (see Section 28.2.3 [Shell Mode], page 250).

Use Lisp mode to run the source files of programs in external Lisps. You can select this mode with *M-x lisp-mode*. It is used automatically for files whose names end in '`.l`' or '`.lisp`', as most Lisp systems usually expect.

When you edit a function in a Lisp program you are running, the easiest way to send the changed definition to the inferior Lisp process is the key *C-M-x*. In Lisp mode, this key runs the function `lisp-send-defun`, which finds the defun around or following point and sends it as input to the Lisp process. (Emacs can send input to any inferior process regardless of what buffer is current.)

Contrast the meanings of *C-M-x* in Lisp mode (for editing programs to be run in another Lisp system) and Emacs-Lisp mode (for editing Lisp programs to be run in Emacs): in both modes it has the effect of installing the function definition that point is in, but the way of doing so is different according to where the relevant Lisp environment is found. See Section 23.2 [Lisp Modes], page 190.

23.8 Packages

The XEmacs 21 distribution comes only with a very basic set of built-in modes and packages. Most of the packages that were part of the distribution of earlier versions of XEmacs are now separately available. The installer as well as the user can choose which packages to install; the actual installation process is easy. This gives an installer the ability to tailor an XEmacs installation for local needs with safe removal of unnecessary code.

23.8.1 Package Flavors

There are two main flavors of packages.

- Regular Packages A regular package is one in which multiple files are involved and one may not in general safely remove any of them.
- Single-File Packages A single-file package is an aggregate collection of thematically related but otherwise independent lisp files. These files are bundled together for download convenience and individual files may deleted at will without any loss of functionality.

23.8.2 Package Distributions

XEmacs Lisp packages are distributed in two ways depending on the intended use. Binary Packages are for installers and end-users and may be installed directly into an XEmacs package directory. Source Packages are for developers and include all files necessary for rebuilding bytecompiled lisp and creating tarballs for distribution.

23.8.3 Binary Packages

Binary packages may be installed directly into an XEmacs package hierarchy.

23.8.4 Source Packages

Source packages contain all of the Package author's (where appropriate in regular packages) source code plus all of the files necessary to build distribution tarballs (Unix Tar format files and gzipped for space savings).

23.8.5 Getting Started

When you first download XEmacs 21, you will usually first grab the *core distribution*, a file called '`xemacs-21.0.tar.gz`'. (Replace the `21.0` by the current version number.) The core distribution contains the sources of XEmacs and a minimal set of Emacs Lisp files, which are in the subdirectory named '`lisp`'. This subdirectory used to contain all Emacs Lisp files distributed with XEmacs. Now, to conserve disk space, most non-essential packages were made optional.

23.8.6 Choosing the Packages You Need

The available packages can currently be found in the same ftp directory where you grabbed the core distribution from, and are located in the subdirectory '`packages/binary-packages`'. Package file names follow the naming convention '`<package-name>-<version>-pkg.tar.gz`'.

If you have EFS ⟨undefined⟩ [(EFS)], page ⟨undefined⟩, packages can be installed over the network. Alternatively, if you have copies of the packages locally, you can install packages from a local disk or CDROM.

The file '`etc/PACKAGES`' in the core distribution contains a list of the packages available at the time of the XEmacs release. Packages are also listed on the `Options` menu under:

```
Options->Customize->Emacs->Packages
```

However, don't select any of these menu picks unless you actually want to install the given package (and have properly configured your system to do so).

You can also get a list of available packages, and whether or not they are installed, using the visual package browser and installer. You can access it via the menus:

```
Options->Manage Packages->List & Install
```

Or, you can get to it via the keyboard:

```
M-x pui-list-packages
```

Hint to system administrators of multi-user systems: it might be a good idea to install all packages and not interfere with the wishes of your users.

23.8.7 XEmacs and Installing Packages

Normally, packages are installed over the network, using EFS ⟨undefined⟩ [(EFS)], page ⟨undefined⟩. However, you may not have network access, or you may already have some or all of the packages on a local disk, such as a CDROM. If you want to install from a local disk, you must first tell XEmacs where to find the package binaries. This is done by adding a line like the following to your '`.emacs`' file:

```
(setq package-get-remote (cons (list nil "/my/path/to/package/binaries")
                               package-get-remote))
```

Here, you'd change '`/my/path/to/package/binaries`' to be the path to your local package binaries. Next, restart XEmacs, and you're ready to go (advanced users can just re-evaluate the sexp).

If you are installing from a temporary, one-time directory, you can also add these directory names to `package-get-remote` using:

```
M-x pui-add-install-directory
```

Note, however, that any directories added using this function are not saved; this information will be lost when you quit XEmacs.

If you're going to install over the network, you only have to insure that EFS ⟨undefined⟩ [(EFS)], page ⟨undefined⟩ works, and that it can get outside a firewall, if you happen to be behind one. You shouldn't have to do anything else; XEmacs already knows where to go. However you can add your own mirrors to this list. See `package-get-remote`.

The easiest way to install a package is to use the visual package browser and installer, using the menu pick:

```
Options->Manage Packages->List & Install
```

or

```
Options->Manage Packages->Using Custom->Select-> ...
```

You can also access it using the keyboard:

```
M-x pui-list-packages
```

The visual package browser will then display a list of all packages. Help information will be displayed at the very bottom of the buffer; you may have to scroll down to see it. You can also press `?` to get the same help. From this buffer, you can tell the package status by the character in the first column:

`-` The package has not been installed.

`*` The package has been installed, but a newer version is available. The current version is out-of-date.

`+` The package has been marked for installation/update.

If there is no character in the first column, the package has been installed and is up-to-date.

From here, you can select or unselect packages for installation using the ⟨RET⟩ key, the *Mouse-2* button or selecting "Select" from the (Popup) Menu. Once you've finished selecting the packages, you can press the *x* key (or use the menu) to actually install the packages. Note that you will have to restart XEmacs for XEmacs to recognize any new packages.

Key summary:

? Display simple help.

⟨RET⟩
⟨Mouse-2⟩ Toggle between selecting and unselecting a package for installation.

x Install selected packages.

⟨SPC⟩ View, in the minibuffer, additional information about the package, such as the package date (not the build date) and the package author. Moving the mouse over a package name will also do the same thing.

v Toggle between verbose and non-verbose package display.

g Refresh the package display.

q Kill the package buffer.

Moving the mouse over a package will also cause additional information about the package to be displayed in the minibuffer.

23.8.8 Other package installation interfaces

For an alternative package interface, you can select packages from the customize menus, under:

```
Options->Customize->Emacs->Packages-> ...
```

or

```
Options->Manage Packages->Using Custom->Select-> ...
```

Set their state to on, and then do:

```
Options->Manage Packages->Using Custom->Update Packages
```

This will automatically retrieve the packages you have selected from the XEmacs ftp site or your local disk, and install them into XEmacs. Additionally it will update any packages you already have installed to the newest version. Note that if a package is newly installed you will have to restart XEmacs for the change to take effect.

You can also install packages using a semi-manual interface:

```
M-x package-get-all <return>
```

Enter the name of the package (e.g., `prog-modes`), and XEmacs will search for the latest version (as listed in the lisp file '`lisp/package-get-base.el`'), and install it and any packages that it depends upon.

23.8.9 Manual Binary Package Installation

Pre-compiled, binary packages can be installed in either a system package directory (this is determined when XEmacs is compiled), or in a subdirectory off your '`$HOME`' directory:

```
~/.xemacs/packages
```

XEmacs does not have to be running to install binary packages, although XEmacs will not know about any newly-installed packages until you restart XEmacs. Note, however, that installing a newer version of a package while XEmacs is running could cause strange errors in XEmacs; it's best to exit XEmacs before upgrading an existing package.

To install binary packages manually:

1. Download the package(s) that you want to install. Each binary package will typically be a gzip'd tarball.
2. Decide where to install the packages: in the system package directory, or in '`~/.xemacs/packages`'. If you want to install the packages in the system package directory, make sure you can write into that directory. If you want to install in your '`$HOME`' directory, create the directory, '`~/.xemacs/packages`'.
3. Next, `cd` to the directory under which you want to install the package(s).
4. From this directory, uncompress and extract each of the gzip'd tarballs that you downloaded in step 1. Unix and Cygnus cygwin users will typically do this using the commands:

   ```
   gunzip < package.tar.gz | tar xvf -
   ```

   Above, replace '`package.tar.gz`' with the filename of the package that you downloaded in step 1.

   Of course, if you use GNU `tar`, you could also use:

   ```
   tar xvzf package.tar.gz
   ```
5. That's it. Quit and restart XEmacs to get it to recognize any new or changed packages.

Source packages are available from the '`packages/source-packages`' subdirectory of your favorite XEmacs distribution site. Alternatively, they are available via CVS from '`cvs.xemacs.org`'. Look at '`http://cvs.xemacs.org`' for instructions.

23.8.10 Prerequisites for Building Source Packages

You must have GNU `cp`, GNU `install` (or a BSD compatible `install` program) GNU `make` (3.75 or later preferred), `makeinfo` (1.68 from `texinfo-3.11` or later required), GNU `tar` and XEmacs 21.0. The source packages will untar into a correct directory structure. At the top level you must have '`XEmacs.rules`' and '`package-compile.el`'. These files are available from the XEmacs FTP site from the same place you obtained your source package distributions.

23.8.11 What You Can Do With Source Packages

NB: A global build operation doesn't exist yet as of 13 January 1998.

Source packages are most useful for creating XEmacs package tarballs for installation into your own XEmacs installations or for distributing to others.

Supported operations from '`make`' are:

`clean`
: Remove all built files except '`auto-autoloads.el`' and '`custom-load.el`'.

`distclean`
: Remove XEmacs backups as well as the files deleted by `make clean`.

`all`
: Bytecompile all files, build and bytecompile byproduct files like '`auto-autoloads.el`' and '`custom-load.el`'. Create info version of TeXinfo documentation if present.

`srckit`
: Usually aliased to `make srckit-std`. This does a `make distclean` and creates a package source tarball in the staging directory. This is generally only of use for package maintainers.

`binkit`
: May be aliased to `binkit-sourceonly`, `binkit-sourceinfo`, `binkit-sourcedata`, or `binkit-sourcedatainfo`. `sourceonly` indicates there is nothing to install in a data directory or info directory. `sourceinfo` indicates that source and info files are to be installed. `sourcedata` indicates that source and etc (data) files are to be installed. `sourcedatainfo` indicates source, etc (data), and info files are to be installed. A few packages have needs beyond the basic templates so this is not yet complete.

`dist`
: Runs the rules `srckit` followed by `binkit`. This is primarily of use by XEmacs maintainers producing files for distribution.

24 Abbrevs

An *abbrev* is a word which *expands* 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 discarded, 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 29.1 [Minor Modes], page 257. `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.

You can define Abbrevs interactively during an editing session. You can also save lists of abbrev definitions in files and reload them 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 206.

24.1 Defining Abbrevs

C-x a *g* Define an abbrev to expand into some text before point (`add-global-abbrev`).

C-x a *l* Similar, but define an abbrev available only in 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, no abbrev definitions remain 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 in the example above, insert the text '`find outer otter`', 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 way they do for *C-x* `a` *g*.

If the text of an 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 existing abbrev definitions.

24.2 Controlling Abbrev Expansion

An abbrev expands whenever it is in a buffer just before point and you type a self-inserting punctuation character (⟨SPC⟩, comma, etc.). Most often an abbrev is used by inserting 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).

Two commands are available 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 may want to use it to enter '`reconstruction`'. It does not work to type *recnst*, because that is not necessarily a defined abbrev. Instead, you can use the command *M-'* (`abbrev-prefix-mark`) 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 by *M-'*. The resulting text is the desired '`reconstruction`'.

If you actually want the text of the abbrev in the buffer, rather than its expansion, insert 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*. You can also use *C-_* (`undo`) to undo the expansion; but that will first undo the insertion of the punctuation character.

M-x expand-region-abbrevs searches through the region for defined abbrevs, and offers to replace each one it finds with its expansion. This command is useful if you have typed 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. The command is effective even if Abbrev mode is not enabled.

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 non-blank 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 want 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 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 `edit-abbrevs-redefine` command does this. Any abbrevs not described in the buffer are eliminated when this is done.

`edit-abbrevs` is actually the same as `list-abbrevs`, except that it selects the buffer '`*Abbrevs*`' whereas `list-abbrevs` merely displays it in another window.

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 an abbrev file and define abbrevs as specified there.

M-x quietly-read-abbrev-file
: Similar, but do not display a message about what is going on.

M-x define-abbrevs
: Define abbrevs from buffer.

M-x insert-abbrevs
: Insert all abbrevs and their expansions into the buffer.

Use *M-x write-abbrev-file* to save abbrev definitions for use in a later session. The command reads a file name using the minibuffer and writes a description of all current abbrev definitions into the specified file. The text stored in the file looks like the output of *M-x list-abbrevs*.

M-x read-abbrev-file prompts for a file name using the minibuffer and reads the specified file, defining abbrevs according to its contents. *M-x quietly-read-abbrev-file* is the same but does not display a message in the echo area; it is actually useful primarily in the '`.emacs`' file. If you give an empty argument to either of these functions, the file name Emacs uses is the value of the variable `abbrev-file-name`, which is by default `"~/.abbrev_defs"`.

Emacs offers 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*). Set the variable `save-abbrevs` to `nil` to inhibit this feature.

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 `edit-picture` to enter Picture mode.

In Picture mode, editing is based on the *quarter-plane* model of text. In this model, 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 non-blank 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. You can then use the *C-c C-c* command (`picture-mode-exit`) to restore that mode. *C-c C-c* also deletes spaces from the ends of lines, unless you give it a numeric argument.

The commands used in Picture mode all work in other modes (provided the '`picture`' library is loaded), but are only bound to keys in Picture mode. Note that the descriptions below talk of moving "one column" and so on, but all the picture mode commands handle numeric arguments as their normal equivalents do.

Turning on Picture mode calls the value of the variable `picture-mode-hook` as a function, with no arguments, if that value exists and is non-`nil`.

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`, which moves point one column to the right, by inserting a space if necessary, so that the actual end of the line makes no difference. *C-b* is rebound to run `picture-backward-column`, which always moves point left one column, converting a tab to multiple spaces if necessary. *C-n* and *C-p* are rebound to run `picture-move-down` and `picture-move-up`, which can either insert spaces or convert tabs as necessary to make sure that point stays in exactly the same column. *C-e* runs `picture-end-of-line`, which moves to after the last non-blank character on the line. There was 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 29.1 [Minor Modes], page 257). 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.

Text is erased instead of deleted and killed. ⟨DEL⟩ (`picture-backward-clear-column`) replaces the preceding character with a space rather than removing it. *C-d* (`picture-clear-column`) does the same in a forward direction. *C-k* (`picture-clear-line`) really kills the contents of lines, but never removes the newlines from a buffer.

To do actual insertion, you must use special commands. *C-o* (`picture-open-line`) 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 remains unchanged. ⟨LFD⟩ (`picture-duplicate-line`) inserts another line with the same contents below the current line.

To actually delete parts of the picture, use *C-w*, or with *C-c C-d* (which is defined as `delete-char`, as *C-d* is in other modes), or with one of the picture rectangle commands (see Section 25.4 [Rectangles in Picture], page 209).

25.2 Controlling Motion After Insert

Since "self-inserting" characters just overwrite and move point in Picture mode, 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. Context-based tabbing is done with *M-*⟨TAB⟩ (`picture-tab-search`). With no argument, it moves to a point underneath the next "interesting" character that follows whitespace in the previous non-blank 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⟩, the 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 of characters 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`. Without arguments 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. If you use this command with ⟨TAB⟩, you 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 10.5 [Rectangles], page 77).

C-c C-k Clear out the region-rectangle (`picture-clear-rectangle`). With argument, kill it.

C-c C-w r Similar but save rectangle contents in register *r* first (`picture-clear-rectangle-to-register`).

C-c C-y Copy last killed rectangle into the buffer by overwriting, with upper left corner at point (`picture-yank-rectangle`). With argument, insert instead.

C-c C-x r Similar, but use the rectangle in register *r* (`picture-yank-rectangle-from-register`).

The picture rectangle commands *C-c C-k* (`picture-clear-rectangle`) and *C-c C-w* (`picture-clear-rectangle-to-register`) differ from the standard rectangle commands in that they normally clear the rectangle instead of deleting it; this is analogous with the way *C-d* is changed in Picture mode.

However, deletion of rectangles can be useful in Picture mode, so these commands delete the rectangle if given a numeric argument.

The Picture mode commands for yanking rectangles differ from the standard ones in overwriting instead of inserting. This is the same way that Picture mode insertion of other text is different from other modes. *C-c C-y* (`picture-yank-rectangle`) inserts (by overwriting) the rectangle that was most recently killed, while *C-c C-x* (`picture-yank-rectangle-from-register`) does for the rectangle found in a specified register.

Since most region commands in Picture mode operate on rectangles, when you select a region of text with the mouse in Picture mode, it is highlighted as a rectangle.

26 Sending Mail

To send a message in Emacs, start by typing the command (*C-x m*) to select and initialize the '`*mail*`' buffer. You can then edit the text and headers of the message in the mail buffer, and type the command (*C-c C-c*) to send the message.

C-x m Begin composing a message to send (`mail`).

C-x 4 m Likewise, but display the message in another window (`mail-other-window`).

C-c C-c In Mail mode, send the message and switch to another buffer (`mail-send-and-exit`).

The command *C-x m* (`mail`) selects a buffer named '`*mail*`' and initializes it with the skeleton of an outgoing message. *C-x 4 m* (`mail-other-window`) selects the '`*mail*`' buffer in a different window, leaving the previous current buffer visible.

Because the buffer for mail composition 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, a new mail buffer will be created; in this way, you can compose multiple messages at once. You can switch back to and complete an unsent message by using the normal buffer selection mechanisms.

C-u C-x m is another way to switch back to a message in progress: it will search for an existing, unsent mail message buffer and select it.

26.1 The Format of the Mail Buffer

In addition to the *text* or contents, a message has *header fields*, which say who sent it, when, to whom, why, and so on. Some header fields, such as the date and sender, are created automatically after the message is sent. Others, such as the recipient names, must be specified by you in order to send the message properly.

Mail mode provides a few commands to help you edit some header fields, and some are preinitialized in the buffer automatically at times. You can insert or edit any header fields using ordinary editing commands.

The line in the buffer that says:

```
--text follows this line--
```

is a special delimiter that separates the headers you have specified from the text. Whatever follows this line is the text of the message; the headers precede it. The delimiter line itself does not appear in the message actually sent. The text used for the delimiter line is controlled by the variable `mail-header-separator`.

Here is an example of what the headers and text in the '`*mail*`' buffer might look like.

```
To: rms@mc
CC: mly@mc, rg@oz
```

```
Subject: The XEmacs User's 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. It does not matter whether you use upper or lower case in the field name. After the colon and optional whitespace comes the contents of the field.

'To' This field contains the mailing addresses of the message.

'Subject' The contents of the 'Subject' field should be a piece of text that says what the message is about. Subject fields are useful because most mail-reading programs can provide a summary of messages, listing the subject of each message but not its text.

'CC' This field contains additional mailing addresses to send the message to, but whose readers should not regard the message as addressed to them.

'BCC' This field contains additional mailing addresses to send the message to, but which should not appear in the header of the message actually sent.

'FCC' This field contains the name of one file (in Unix mail file format) to which a copy of the message should be appended when the message is sent.

'From' Use the 'From' field to say who you are, when the account you are using to send the mail is not your own. The contents of the 'From' field should be a valid mailing address, since replies will normally go there.

'Reply-To'
Use the 'Reply-To' field to direct replies to a different address, not your own. 'From' and 'Reply-To' have the same effect on where replies go, but they convey a different meaning to the person who reads the message.

'In-Reply-To'
This field contains a piece of text describing a message you are replying to. Some mail systems can use the information to correlate related pieces of mail. This field is normally filled in by your mail handling package when you are replying to a message and you never need to think about it.

The 'To', 'CC', 'BCC' and 'FCC' fields can appear any number of times, to specify many places to send the message.

The 'To', 'CC', and 'BCC', fields can have continuation lines. All the lines starting with whitespace, following the line on which the field starts, are considered part of the field. For example,

```
To: foo@here, this@there,
  me@gnu.cambridge.mass.usa.earth.spiral3281
```

If you have a '~/.mailrc' file, Emacs scans it for mail aliases the first time you try to send mail in an Emacs session. Emacs expands aliases found in the 'To', 'CC', and 'BCC' fields where appropriate. You can set the variable mail-abbrev-mailrc-file to the name of the file with mail aliases. If nil, '~/.mailrc' is used.

Your '`.mailrc`' file ensures that word-abbrevs are defined for each of your mail aliases when point is in a '`To`', '`CC`', '`BCC`', or '`From`' field. The aliases are defined in your '`.mailrc`' file or in a file specified by the **MAILRC** environment variable if it exists. Your mail aliases expand any time you type a word-delimiter at the end of an abbreviation.

In this version of Emacs, what you see is what you get: in contrast to some other versions, no abbreviations are expanded after you have sent the mail. This means you don't suffer the annoyance of having the system do things behind your back — if the system rewrites an address you typed, you know it immediately, instead of after the mail has been sent and it's too late to do anything about it. For example, you will never again be in trouble because you forgot to delete an old alias from your '`.mailrc`' and a new local user is given a userid which conflicts with one of your aliases.

Your mail alias abbrevs are in effect only when point is in an appropriate header field. The mail aliases will not expand in the body of the message, or in other header fields. The default mode-specific abbrev table `mail-mode-abbrev-table` is used instead if defined. That means if you have been using mail-mode specific abbrevs, this code will not adversely affect you. You can control which header fields the abbrevs are used in by changing the variable `mail-abbrev-mode-regexp`.

If auto-fill mode is on, abbrevs wrap at commas instead of at word boundaries, and header continuation lines will be properly indented.

You can also insert a mail alias with `mail-interactive-insert-alias`. This function, which is bound to *C-c C-a*, prompts you for an alias (with completion) and inserts its expansion at point.

In this version of Emacs, it is possible to have lines like the following in your '`.mailrc`' file:

```
alias someone "John Doe <doe@quux.com>"
```

That is, if you want an address to have embedded spaces, simply surround it with double-quotes. The quotes are necessary because the format of the '`.mailrc`' file uses spaces as address delimiters.

Aliases in the '`.mailrc`' file may be nested. For example, assume you define aliases like:

```
alias group1 fred ethel
alias group2 larry curly moe
alias everybody group1 group2
```

When you now type '`everybody`' on the '`To`' line, it will expand to:

```
fred, ethyl, larry, curly, moe
```

Aliases may contain forward references; the alias of '`everybody`' in the example above can precede the aliases of '`group1`' and '`group2`'.

In this version of Emacs, you can use the `source` '`.mailrc`' command for reading aliases from some other file as well.

Aliases may contain hyphens, as in `"alias foo-bar foo@bar"`, even though word-abbrevs normally cannot contain hyphens.

To read in the contents of another '.mailrc'-type file from Emacs, use the command `M-x merge-mail-aliases`. The `rebuild-mail-aliases` command is similar, but deletes existing aliases first.

If you want multiple addresses separated by a string other than ',' (a comma), then set the variable `mail-alias-seperator-string` to it. This has to be a comma bracketed by whitespace if you want any kind of reasonable behavior.

If the variable `mail-archive-file-name` is non-`nil`, it should be a string naming a file. Each time you start to edit a message to send, an 'FCC' field is entered for that file. Unless you remove the 'FCC' field, every message is written into that file when it is sent.

26.3 Mail Mode

The major mode used in the '*mail*' buffer is Mail mode. Mail mode is similar to Text mode, but several commands are provided on the *C-c* prefix. These commands all deal 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 (`mail-yank-original`).

C-c C-q Fill all paragraphs of yanked old messages, each individually (`mail-fill-yanked-message`).

⟨button3⟩ Pops up a menu of useful mail-mode commands.

There are two ways to send a message. *C-c C-c* (`mail-send-and-exit`) is the usual way to send the message. It sends the message and then deletes the window (if there is another window) or switches to another buffer. It puts the '*mail*' buffer at the lowest priority for automatic reselection, since you are finished with using it. *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.

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 edited most frequently.

C-c C-w (`mail-signature`) adds a standard piece of text at the end of the message to say more about who you are. The text comes from the file '.signature' in your home directory.

When you use an Rmail command to send mail from the Rmail mail reader, you can use *C-c C-y* `mail-yank-original` inside the '`*mail*`' buffer to insert the text of the message you are replying to. Normally Rmail indents each line of that message four spaces and eliminates most header fields. A numeric argument specifies the number of spaces to indent. An argument of just *C-u* says not to indent at all and not to eliminate anything. *C-c C-y* always uses the current message from the '`RMAIL`' buffer, so you can insert several old messages by selecting one in '`RMAIL`', switching to '`*mail*`' and yanking it, then switching back to '`RMAIL`' to select another.

After using *C-c C-y*, 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 separately.

Clicking the right mouse button in a mail buffer pops up a menu of the above commands, for easy access.

Turning on Mail mode (which *C-x m* does automatically) calls the value of `text-mode-hook`, if it is not void or `nil`, and then calls the value of `mail-mode-hook` if that is not void or `nil`.

27 Reading Mail

XEmacs provides three separate mail-reading packages. Each one comes with its own manual, which is included standard with the XEmacs distribution.

The recommended mail-reading package for new users is VM. VM works with standard Unix-mail-format folders and was designed as a replacement for the older Rmail.

XEmacs also provides a sophisticated and comfortable front-end to the MH mail-processing system, called '`mh-e`'. Unlike in other mail programs, folders in MH are stored as file-system directories, with each message occupying one (numbered) file. This facilitates working with mail using shell commands, and many other features of MH are also designed to integrate well with the shell and with shell scripts. Keep in mind, however, that in order to use mh-e you must have the MH mail-processing system installed on your computer.

Finally, XEmacs provides the Rmail package. Rmail is (currently) the only mail reading package distributed with FSF GNU Emacs, and is powerful in its own right. However, it stores mail folders in a special format called '`Babyl`', that is incompatible with all other frequently-used mail programs. A utility program is provided for converting Babyl folders to standard Unix-mail format; however, unless you already have mail in Babyl-format folders, you should consider using VM or mh-e instead. (If at times you have to use FSF Emacs, it is not hard to obtain and install VM for that editor.)

27.1 Calendar Mode 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. With a numeric 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, whose major mode is Calendar mode.

`Button2` in the calendar brings up a menu of operations on a particular date; *`Buttons3`* brings up a menu of commonly used calendar features that are independent of any particular date. To exit the calendar, type *`q`*. See section "Calendar" in *The Emacs Lisp Reference Manual*, for customization information about the calendar and diary.

27.1.1 Movement in the Calendar

Calendar mode lets you move through the calendar 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 to make the selected date visible. Moving to a date lets you view its holidays or diary entries, or convert it to other calendars; moving longer time periods is also useful simply to scroll the calendar.

27.1.1.1 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-backward-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.

27.1.1.2 Beginning or End of Week, Month or Year

A week (or month, or year) is not just a quantity of days; we think of weeks (months, years) as starting on particular dates. So Calendar mode provides commands to move to the beginning or end of a week, month or year:

C-a Move point to start of week (`calendar-beginning-of-week`).

C-e Move point to end of week (`calendar-end-of-week`).

M-a Move point to start of month (`calendar-beginning-of-month`).

M-e Move point to end of month (`calendar-end-of-month`).

M-< Move point to start 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.

27.1.1.3 Particular Dates

Calendar mode provides commands for moving to a particular date specified in various ways.

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-goto-today`).

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-goto-today`).

27.1.2 Scrolling the Calendar through Time

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`).

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` 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.

27.1.3 The Mark and the Region

The concept of the mark applies to the calendar just as to any other buffer, but it marks a *date*, not a *position* in the buffer. The region consists of the days between the mark and point (including the starting and stopping dates).

`C-SPC` Set the mark to today's date (`calendar-set-mark`).

`C-@` The same.

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

`M-=` Display the number of days in the current region (`calendar-count-days-region`).

You set the mark in the calendar, as in any other buffer, by using `C-@` or `C-SPC` (`calendar-set-mark`). You return to the marked date with the command `C-x C-x` (`calendar-exchange-point-and-mark`) which puts the mark where point was and point where mark was. The calendar is scrolled as necessary, if the marked date was not visible on the screen. This does not change the extent of the 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.

The main use of the mark in the calendar is to remember dates that you may want to go back to. To make this feature more useful, the mark ring (see Section 9.1.4 [Mark Ring], page 65) operates exactly as in other buffers: Emacs remembers 16 previous locations of the mark. To return to a marked date, type `C-u C-SPC` (or `C-u C-@`); this is the command `calendar-set-mark` given a numeric argument. It moves point to where the mark was, restores the mark from the ring of former marks, and stores the previous point at the end of the mark ring. So, repeated use of this command moves point through all the old marks on the ring, one by one.

27.1.4 Miscellaneous Calendar Commands

`p d` Display day-in-year (`calendar-print-day-of-year`).

`?` Briefly describe calendar commands (`describe-calendar-mode`).

`C-c C-l` Regenerate the calendar window (`redraw-calendar`).

`SPC` Scroll the next window (`scroll-other-window`).

`q` Exit from calendar (`exit-calendar`).

If you want to know how many days have 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.

To display a brief description of the calendar commands, type `?` (`describe-calendar-mode`). For a fuller description, type `C-h m`.

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.

If the calendar window text gets corrupted, type `C-c C-l` (`redraw-calendar`) to redraw it. (This can only happen if you use non-Calendar-mode editing commands.)

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.

To exit from the calendar, type `q` (`exit-calendar`). This buries all buffers related to the calendar, selecting other buffers. (If a frame contains a dedicated calendar window, exiting from the calendar iconifies that frame.)

27.2 LaTeX Calendar

The Calendar LaTeX commands produce a buffer of LaTeX code that prints as a calendar. Depending on the command you use, the printed calendar covers the day, week, month or year that point is in.

`t m` Generate a one-month calendar (`cal-tex-cursor-month`).

`t M` Generate a sideways-printing one-month calendar (`cal-tex-cursor-month-landscape`).

`t d` Generate a one-day calendar (`cal-tex-cursor-day`).

`t w 1` Generate a one-page calendar for one week (`cal-tex-cursor-week`).

`t w 2` Generate a two-page calendar for one week (`cal-tex-cursor-week2`).

`t w 3` Generate an ISO-style calendar for one week (`cal-tex-cursor-week-iso`).

`t w 4` Generate a calendar for one Monday-starting week (`cal-tex-cursor-week-monday`).

`t f w` Generate a Filofax-style two-weeks-at-a-glance calendar (`cal-tex-cursor-filofax-2week`).

`t f W` Generate a Filofax-style one-week-at-a-glance calendar (`cal-tex-cursor-filofax-week`).

`t y` Generate a calendar for one year (`cal-tex-cursor-year`).

`t Y` Generate a sideways-printing calendar for one year (`cal-tex-cursor-year-landscape`).

`t f y` Generate a Filofax-style calendar for one year (`cal-tex-cursor-filofax-year`).

Some of these commands print the calendar sideways (in "landscape mode"), so it can be wider than it is long. Some of them use Filofax paper size (3.75in x 6.75in). All of these commands accept a prefix argument which specifies how many days, weeks, months or years to print (starting always with the selected one).

If the variable `cal-tex-holidays` is non-`nil` (the default), then the printed calendars show the holidays in `calendar-holidays`. If the variable `cal-tex-diary` is non-`nil` (the default is `nil`), diary entries are included also (in weekly and monthly calendars only).

27.2.1 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`).

Button2 Holidays
Display any holidays for the date you click on.

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.

M-x list-holidays
List holidays in another window for a specified range of years.

To see if any holidays fall on a given date, position point on that date in the calendar window and use the *h* command. Alternatively, click on that date with *Button2* and then choose *Holidays* from the menu that appears. Either way, this displays the holidays for that date, in the echo area if they fit there, otherwise in a separate window.

To view the distribution of holidays for all the dates shown in the calendar, use the *x* command. This displays the dates that are holidays in a different face (or places a '`*`' after these dates, if display with multiple faces is not available). 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 27.5.2 [Diary], page 228).

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. You can use ⟨SPC⟩ in the calendar window to scroll that list.

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 United States holidays and the major Christian, Jewish, and Islamic holidays; also the solstices and equinoxes.

The command *M-x list-holidays* displays the list of holidays for a range of years. This function asks you for the starting and stopping years, and allows you to choose all the holidays or one of several categories of holidays. You can use this command even if you don't have a calendar window.

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 United States law mandates that daylight savings time begins on the first Sunday in April. When the daylight savings rules are set up for the United States, Emacs always uses the present definition, even though it is wrong for some prior years.

27.2.2 Times of Sunrise and Sunset

Special calendar commands can tell you, to within a minute or two, the times of sunrise and sunset for any date.

S Display times of sunrise and sunset for the selected date (`calendar-sunrise-sunset`).

Button2 Sunrise/Sunset
: Display times of sunrise and sunset for the date you click on.

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*. Alternatively, click *Button2* on the date, then choose *Sunrise/Sunset* from the menu that appears. The command *M-x sunrise-sunset* is available outside the calendar to display 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.

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.

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 time zone 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 Section 27.5.8.5 [Daylight Savings], page 239, for how daylight savings time is determined.

As a user, you might find it convenient to set the calendar location variables for your usual physical location in your '`.emacs`' file. And when you install Emacs on a machine, you can create a '`default.el`' file which sets them properly for the typical location of most users of that machine. See Section 29.6 [Init File], page 277.

27.2.3 Phases of the Moon

These calendar commands display the dates and times of the phases of the moon (new moon, first quarter, full moon, last quarter). 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 Section 27.5.8.5 [Daylight Savings], page 239.

27.2.4 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.

If you are interested in these calendars, you can convert dates one at a time. Put point on the desired date of the Gregorian calendar and press the appropriate keys. The *`p`* is a mnemonic for "print" since Emacs "prints' the equivalent date in the echo area.

27.3 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 used by tradition in the Jewish religion. The Emacs calendar program uses the Hebrew calendar to determine the dates of Jewish holidays. Hebrew calendar dates begin and end at sunset.

The Islamic calendar is used in many predominantly Islamic countries. Emacs uses it to determine the dates of Islamic 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 observance 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.

The Copts use a calendar based on the ancient Egyptian solar calendar. Their calendar consists of twelve 30-day months followed by an extra five-day period. Once every fourth year they add a leap day to this extra period to make it six days. The Ethiopic calendar is identical in structure, but has different year numbers and month names.

The Persians use a solar calendar based on a design of Omar Khayyam. Their calendar consists of twelve months of which the first six have 31 days, the next five have 30 days, and the last has 29 in ordinary years and 30 in leap years. Leap years occur in a complicated pattern every four or five years.

The Chinese calendar is a complicated system of lunar months arranged into solar years. The years go in cycles of sixty, each year containing either twelve months in an ordinary year or thirteen months in a leap year; each month has either 29 or 30 days. Years, ordinary months, and days are named by combining one of ten “celestial stems” with one of twelve “terrestrial branches” for a total of sixty names that are repeated in a cycle of sixty.

27.4 Converting To Other Calendars

The following commands describe the selected date (the date at point) in various other calendar systems:

Button2 Other Calendars
: Display the date that you click on, expressed in various other calendars.

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 C
: Display Chinese date for selected day (`calendar-print-chinese-date`).

p k
: Display Coptic date for selected day (`calendar-print-coptic-date`).

p e
: Display Ethiopic date for selected day (`calendar-print-ethiopic-date`).

p p
: Display Persian date for selected day (`calendar-print-persian-date`).

p m
: Display Mayan date for selected day (`calendar-print-mayan-date`).

If you are using X, the easiest way to translate a date into other calendars is to click on it with *Button2*, then choose *Other Calendars* from the menu that appears. This displays the equivalent forms of the date in all the calendars Emacs understands, in the form of a menu. (Choosing an alternative from this menu doesn't actually do anything—the menu is used only for display.)

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.

27.5 Converting From Other Calendars

You can use the other supported calendars to specify a date to move to. This section describes the commands for doing this using calendars other than Mayan; for the Mayan calendar, see the following section.

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`).

g C
: Move to a date specified in the Chinese calendar (`calendar-goto-chinese-date`).

g p
: Move to a date specified in the Persian calendar (`calendar-goto-persian-date`).

g k
: Move to a date specified in the Coptic calendar (`calendar-goto-coptic-date`).

g e Move to a date specified in the Ethiopic calendar (`calendar-goto-ethiopic-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 53) 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.

27.5.1 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 n t Move to the next occurrence of a place in the tzolkin calendar (`calendar-next-tzolkin-date`).

g m p t Move to the previous occurrence of a place in the tzolkin calendar (`calendar-previous-tzolkin-date`).

g m n h Move to the next occurrence of a place in the haab calendar (`calendar-next-haab-date`).

g m p h Move to the previous occurrence of a place in the haab calendar (`calendar-previous-haab-date`).

g m n c Move to the next occurrence of a place in the calendar round (`calendar-next-calendar-round-date`).

g m p c Move to the previous occurrence of a place in the calendar round (`calendar-previous-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 n c* to move point to the next occurrence of a combination. These commands signal an error if the haab/tzolkin date combination you have typed is impossible.

Emacs uses strict completion (see Section 6.3 [Completion], page 53) whenever it asks you to type a Mayan name, so you don't have to worry about spelling.

27.5.2 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.
```

This example uses extra spaces to align the event descriptions of most of the entries. Such formatting is purely a matter of taste.

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. You can also share diary entries with other users (see Section 27.5.8.9 [Included Diary Files], page 243).

27.5.3 Commands Displaying Diary Entries

Once you have created a '`~/diary`' file, you can use the calendar to view it. You can also view today's events outside of Calendar mode.

`d` Display all diary entries for the selected date (`view-diary-entries`).

`Button2 Diary`
Display all diary entries for the date you click on.

`s` Display the entire diary file (`show-all-diary-entries`).

`m` Mark all visible dates that have diary entries (`mark-diary-entries`).

`u` Unmark the calendar window (`calendar-unmark`).

`M-x print-diary-entries`
Print hard copy of the diary display as it appears.

`M-x diary` Display all diary entries for today's date.

`M-x diary-mail-entries`
Mail yourself email reminders about upcoming diary entries.

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.

Another way to display the diary entries for a date is to click *`Button2`* on the date, and then choose *`Diary`* from the menu that appears.

To get a broader view of which days are mentioned in the diary, use the *`m`* command. This displays the dates that have diary entries in a different face (or places a '`+`' after these dates, if display with multiple faces is not available). 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 27.2.1 [Holidays], page 222).

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.

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 28.4 [Hardcopy], page 253).

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 Chapter 29 [Customization], page 257).

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.

Many users like to receive notice of events in their diary as email. To send such mail to yourself, use the command *`M-x diary-mail-entries`*. A prefix argument specifies how many days (starting with today) to check; otherwise, the variable `diary-mail-days` says how many days.

27.5.4 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. The `calendar` utility program supports a subset of the format allowed by the Emacs diary facilities, so you can use that utility to view the diary file, with reasonable results aside from the entries it cannot understand.

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, this entry:

```
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`).

27.5.5 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. The optional *year* is also 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*; that is, 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.

27.5.6 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 by default. To make a nonmarking diary entry, give a numeric 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.

27.5.7 Special Diary Entries

In addition to entries based on calendar dates, the diary file can contain *sexp 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 sexp 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 sexp 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:

The effect of *i a* is to add a `diary-anniversary` sexp to your diary file. You can also add one manually, for instance:

```
%%(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.

A *block* diary entry applies to a specified range of consecutive dates. 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.)

To insert a block entry, place point and the mark on the two dates that begin and end the range, and type `i b`. This command displays the end of your diary file in another window and inserts the block description; you can then type the diary entry.

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, which looks like this:

```
%%(diary-cyclic 50 3 1 1990) Renew medication
```

This entry 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 these commands make marking diary entries. To insert a nonmarking entry, 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.

The sexp feature of the diary allows you to specify diary entries based on any Emacs Lisp expression. You can use the library of built-in functions or you can write your own functions. The built-in functions include the ones shown in this section, plus a few others (see Section 27.5.8.10 [Sexp Diary Entries], page 243).

The generality of sexps lets you specify any diary entry that you can describe algorithmically. Suppose you get paid on the 21st of the month if it is a weekday, and to the Friday before if the 21st is on a weekend. The diary entry

```
&%%(let ((dayname (calendar-day-of-week date))
         (day (car (cdr date))))
      (or (and (= day 21) (memq dayname '(1 2 3 4 5)))
          (and (memq day '(19 20)) (= dayname 5)))
         ) Pay check deposited
```

to just those dates. This example illustrates how the sexp can depend on the variable `date`; this variable is a list (*month day year*) that gives the Gregorian date for which the diary entries are being found. If the value of the sexp is `t`, the entry applies to that date. If the sexp evaluates to `nil`, the entry does *not* apply to that date.

27.5.8 Customizing the Calendar and Diary

There are many customizations that you can use to make the calendar and diary suit your personal tastes.

27.5.8.1 Customizing the Calendar

If you set the variable `view-diary-entries-initially` to `t`, calling up the calendar automatically displays the diary entries for the current date as well. The diary dates appear only if the current date is visible. If you add both of the following lines to your '`.emacs`' file:

```
(setq view-diary-entries-initially t)
(calendar)
```

this displays both the calendar and diary windows whenever you start Emacs.

Similarly, if you set the variable `view-calendar-holidays-initially` to `t`, entering the calendar automatically displays a list of holidays for the current three-month period. The holiday list appears in a separate window.

You can set the variable `mark-diary-entries-in-calendar` to `t` in order to mark any dates with diary entries. This takes effect whenever the calendar window contents are recomputed. There are two ways of marking these dates: by changing the face (see Section 29.8 [Faces], page 281), if the display supports that, or by placing a plus sign ('`+`') beside the date otherwise.

Similarly, setting the variable `mark-holidays-in-calendar` to `t` marks holiday dates, either with a change of face or with an asterisk ('`*`').

The variable `calendar-holiday-marker` specifies how to mark a date as being a holiday. Its value may be a character to insert next to the date, or a face name to use for displaying the date.

Likewise, the variable `diary-entry-marker` specifies how to mark a date that has diary entries. The calendar creates faces named `holiday-face` and `diary-face` for these purposes; those symbols are the default values of these variables, when Emacs supports multiple faces on your terminal.

The variable `calendar-load-hook` is a normal hook run when the calendar package is first loaded (before actually starting to display the calendar).

Starting the calendar runs the normal hook `initial-calendar-window-hook`. Recomputation of the calendar display does not run this hook. But if you leave the calendar with the `q` command and reenter it, the hook runs again.

The variable `today-visible-calendar-hook` is a normal hook run after the calendar buffer has been prepared with the calendar when the current date is visible in the window. One use of this hook is to replace today's date with asterisks; to do that, use the hook function `calendar-star-date`.

```
(add-hook 'today-visible-calendar-hook 'calendar-star-date)
```

Another standard hook function marks the current date, either by changing its face or by adding an asterisk. Here's how to use it:

```
(add-hook 'today-visible-calendar-hook 'calendar-mark-today)
```

The variable `calendar-today-marker` specifies how to mark today's date. Its value should be a character to insert next to the date or a face name to use for displaying the date. A face named `calendar-today-face` is provided for this purpose; that symbol is the default for this variable when Emacs supports multiple faces on your terminal.

A similar normal hook, `today-invisible-calendar-hook` is run if the current date is *not* visible in the window.

27.5.8.2 Customizing the Holidays

Emacs knows about holidays defined by entries on one of several lists. You can customize these lists of holidays to your own needs, adding or deleting holidays. The lists of holidays that Emacs uses are for general holidays (`general-holidays`), local holidays (`local-holidays`), Christian holidays (`christian-holidays`), Hebrew (Jewish) holidays (`hebrew-holidays`), Islamic (Moslem) holidays (`islamic-holidays`), and other holidays (`other-holidays`).

The general holidays are, by default, holidays common throughout the United States. To eliminate these holidays, set `general-holidays` to `nil`.

There are no default local holidays (but sites may supply some). You can set the variable `local-holidays` to any list of holidays, as described below.

By default, Emacs does not include all the holidays of the religions that it knows, only those commonly found in secular calendars. For a more extensive collection of religious holidays, you can set any (or all) of the variables `all-christian-calendar-holidays`, `all-hebrew-calendar-holidays`, or `all-islamic-calendar-holidays` to `t`. If you want to eliminate the religious holidays, set any or all of the corresponding variables `christian-holidays`, `hebrew-holidays`, and `islamic-holidays` to `nil`.

You can set the variable `other-holidays` to any list of holidays. This list, normally empty, is intended for individual use.

Each of the lists (`general-holidays`, `local-holidays`, `christian-holidays`, `hebrew-holidays`, `islamic-holidays`, and `other-holidays`) is a list of *holiday forms*, each holiday form describing a holiday (or sometimes a list of holidays).

Here is a table of the possible kinds of holiday form. Day numbers and month numbers count starting from 1, but "dayname" numbers count Sunday as 0. The element *string* is always the name of the holiday, as a string.

`(holiday-fixed` *month day string*`)`
: A fixed date on the Gregorian calendar. *month* and *day* are numbers, *string* is the name of the holiday.

`(holiday-float` *month dayname k string*`)`
: The *k*th *dayname* in *month* on the Gregorian calendar (*dayname*=0 for Sunday, and so on); negative *k* means count back from the end of the month. *string* is the name of the holiday.

`(holiday-hebrew` *month day string*`)`
: A fixed date on the Hebrew calendar. *month* and *day* are numbers, *string* is the name of the holiday.

`(holiday-islamic` *month day string*`)`
: A fixed date on the Islamic calendar. *month* and *day* are numbers, *string* is the name of the holiday.

`(holiday-julian` *month day string*`)`
: A fixed date on the Julian calendar. *month* and *day* are numbers, *string* is the name of the holiday.

`(holiday-sexp` *sexp string*`)`
: A date calculated by the Lisp expression *sexp*. The expression should use the variable `year` to compute and return the date of a holiday, or `nil` if the holiday doesn't happen this year. The value of *sexp* must represent the date as a list of the form `(`*month day year*`)`. *string* is the name of the holiday.

`(if` *condition holiday-form* `&optional` *holiday-form*`)`
: A holiday that happens only if *condition* is true.

`(`*function* [*args*]`)`
: A list of dates calculated by the function *function*, called with arguments *args*.

For example, suppose you want to add Bastille Day, celebrated in France on July 14. You can do this by adding the following line to your '`.emacs`' file:

```
(setq other-holidays '((holiday-fixed 7 14 "Bastille Day")))
```

The holiday form `(holiday-fixed 7 14 "Bastille Day")` specifies the fourteenth day of the seventh month (July).

Many holidays occur on a specific day of the week, at a specific time of month. Here is a holiday form describing Hurricane Supplication Day, celebrated in the Virgin Islands on the fourth Monday in August:

```
(holiday-float 8 1 4 "Hurricane Supplication Day")
```

Here the 8 specifies August, the 1 specifies Monday (Sunday is 0, Tuesday is 2, and so on), and the 4 specifies the fourth occurrence in the month (1 specifies the first occurrence, 2 the second occurrence, −1 the last occurrence, −2 the second-to-last occurrence, and so on).

You can specify holidays that occur on fixed days of the Hebrew, Islamic, and Julian calendars too. For example,

```
(setq other-holidays
      '((holiday-hebrew 10 2 "Last day of Hanukkah")
        (holiday-islamic 3 12 "Mohammed's Birthday")
        (holiday-julian 4 2 "Jefferson's Birthday")))
```

adds the last day of Hanukkah (since the Hebrew months are numbered with 1 starting from Nisan), the Islamic feast celebrating Mohammed's birthday (since the Islamic months are numbered from 1 starting with Muharram), and Thomas Jefferson's birthday, which is 2 April 1743 on the Julian calendar.

To include a holiday conditionally, use either Emacs Lisp's `if` or the `holiday-sexp` form. For example, American presidential elections occur on the first Tuesday after the first Monday in November of years divisible by 4:

```
(holiday-sexp (if (= 0 (% year 4))
                   (calendar-gregorian-from-absolute
                  (1+ (calendar-dayname-on-or-before
                        1 (+ 6 (calendar-absolute-from-gregorian
                                 (list 11 1 year))))))
              "US Presidential Election"))
```

or

```
(if (= 0 (% displayed-year 4))
    (fixed 11
           (extract-calendar-day
             (calendar-gregorian-from-absolute
               (1+ (calendar-dayname-on-or-before
                     1 (+ 6 (calendar-absolute-from-gregorian
                              (list 11 1 displayed-year)))))))
           "US Presidential Election"))
```

Some holidays just don't fit into any of these forms because special calculations are involved in their determination. In such cases you must write a Lisp function to do the calculation. To include eclipses, for example, add `(eclipses)` to `other-holidays` and write an Emacs Lisp function `eclipses` that returns a (possibly empty) list of the relevant Gregorian dates among the range visible in the calendar window, with descriptive strings, like this:

```
(((6 27 1991) "Lunar Eclipse") ((7 11 1991) "Solar Eclipse") ... )
```

27.5.8.3 Date Display Format

You can customize the manner of displaying dates in the diary, in mode lines, and in messages by setting `calendar-date-display-form`. This variable holds a list of expressions that can involve the variables `month`, `day`, and `year`, which are all numbers in string form, and `monthname` and `dayname`, which are both alphabetic strings. In the American style, the default value of this list is as follows:

```
((if dayname (concat dayname ", ")) monthname " " day ", " year)
```

while in the European style this value is the default:

```
((if dayname (concat dayname ", ")) day " " monthname " " year)
```

+The ISO standard date representation is this:

```
(year "-" month "-" day)
```

This specifies a typical American format:

```
(month "/" day "/" (substring year -2))
```

27.5.8.4 Time Display Format

The calendar and diary by default display times of day in the conventional American style with the hours from 1 through 12, minutes, and either '`am`' or '`pm`'. If you prefer the European style, also known in the US as military, in which the hours go from 00 to 23, you can alter the variable `calendar-time-display-form`. This variable is a list of expressions that can involve the variables `12-hours`, `24-hours`, and `minutes`, which are all numbers in string form, and `am-pm` and `time-zone`, which are both alphabetic strings. The default value of `calendar-time-display-form` is as follows:

```
(12-hours ":" minutes am-pm
          (if time-zone " (") time-zone (if time-zone ")"))
```

Here is a value that provides European style times:

```
(24-hours ":" minutes
          (if time-zone " (") time-zone (if time-zone ")"))
```

gives military-style times like '`21:07 (UT)`' if time zone names are defined, and times like '`21:07`' if they are not.

27.5.8.5 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.

If the default choice of rules is not appropriate for your location, you can tell Emacs the rules to use by setting the variables `calendar-daylight-savings-starts` and `calendar-daylight-savings-ends`. Their 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)
```

That is, 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)
```

For a more complex example, suppose daylight savings time begins on the first of Nisan on the Hebrew calendar. You should set `calendar-daylight-savings-starts` to this value:

```
(calendar-gregorian-from-absolute
  (calendar-absolute-from-hebrew
    (list 1 1 (+ year 3760))))
```

because Nisan is the first month in the Hebrew calendar and the Hebrew year differs from the Gregorian year by 3760 at Nisan.

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`.

The variable `calendar-daylight-time-offset` specifies the difference between daylight savings time and standard time, measured in minutes. The value for Cambridge, Massachusetts is 60.

The two variables `calendar-daylight-savings-starts-time` and `calendar-daylight-savings-ends-time` specify the number of minutes after midnight local time when the transition to and from daylight savings time should occur. For Cambridge, Massachusetts both variables' values are 120.

27.5.8.6 Customizing the Diary

Ordinarily, the mode line of the diary buffer window indicates any holidays that fall on the date of the diary entries. The process of checking for holidays can take several seconds, so including holiday information delays the display of the diary buffer noticeably. If you'd prefer to have a faster display of the diary buffer but without the holiday information, set the variable `holidays-in-diary-buffer` to `nil`.

The variable `number-of-diary-entries` controls the number of days of diary entries to be displayed at one time. It affects the initial display when `view-diary-entries-initially` is `t`, as well as the command *`M-x diary`*. For example, the default value is 1, which says to display only the current day's diary entries. If the value is 2, both the current day's and the next day's entries are displayed. The value can also be a vector of seven elements: for example, if the value is `[0 2 2 2 2 4 1]` then no diary entries appear on Sunday, the current date's and the next day's diary entries appear Monday through Thursday, Friday through Monday's entries appear on Friday, while on Saturday only that day's entries appear.

The variable `print-diary-entries-hook` is a normal hook run after preparation of a temporary buffer containing just the diary entries currently visible in the diary buffer. (The other, irrelevant diary entries are really absent from the temporary buffer; in the diary buffer, they are merely hidden.) The default value of this hook does the printing with the command `lpr-buffer`. If you want to use a different command to do the printing, just change the value of this hook. Other uses might include, for example, rearranging the lines into order by day and time.

You can customize the form of dates in your diary file, if neither the standard American nor European styles suits your needs, by setting the variable `diary-date-forms`. This variable is a list of patterns for recognizing a date. Each date pattern is a list whose elements may be regular expressions (see Section 13.5 [Regexps], page 89) or the symbols `month`, `day`, `year`, `monthname`, and `dayname`. All these elements serve as patterns that match certain kinds of text in the diary file. In order for the date pattern, as a whole, to match, all of its elements must match consecutively.

A regular expression in a date pattern matches in its usual fashion, using the standard syntax table altered so that '`*`' is a word constituent.

The symbols `month`, `day`, `year`, `monthname`, and `dayname` match the month number, day number, year number, month name, and day name of the date being considered. The symbols that match numbers allow leading zeros; those that match names allow three-letter abbreviations and capitalization. All the symbols can match '`*`'; since '`*`' in a diary entry means "any day", "any month", and so on, it should match regardless of the date being considered.

The default value of `diary-date-forms` in the American style is this:

```
((month "/" day "[^/0-9]")
 (month "/" day "/" year "[^0-9]")
 (monthname " *" day "[^,0-9]")
```

```
(monthname " *" day ", *" year "[^0-9]")
(dayname "\\W"))
```

Emacs matches of the diary entries with the date forms is done with the standard syntax table from Fundamental mode (see section "Syntax Tables" in *XEmacs Lisp Reference Manual*), but with the '`*`' changed so that it is a word constituent.

The date patterns in the list must be *mutually exclusive* and must not match any portion of the diary entry itself, just the date and one character of whitespace. If, to be mutually exclusive, the pattern must match a portion of the diary entry text—beyond the whitespace that ends the date—then the first element of the date pattern *must* be `backup`. This causes the date recognizer to back up to the beginning of the current word of the diary entry, after finishing the match. Even if you use `backup`, the date pattern must absolutely not match more than a portion of the first word of the diary entry. The default value of `diary-date-forms` in the European style is this list:

```
((day "/" month "[^/0-9]")
 (day "/" month "/" year "[^0-9]")
 (backup day " *" monthname "\\W+\\<[^*0-9]")
 (day " *" monthname " *" year "[^0-9]")
 (dayname "\\W"))
```

Notice the use of `backup` in the third pattern, because it needs to match part of a word beyond the date itself to distinguish it from the fourth pattern.

27.5.8.7 Hebrew- and Islamic-Date Diary Entries

Your diary file can have entries based on Hebrew or Islamic dates, as well as entries based on the world-standard Gregorian calendar. However, because recognition of such entries is time-consuming and most people don't use them, you must explicitly enable their use. If you want the diary to recognize Hebrew-date diary entries, for example, you must do this:

```
(add-hook 'nongregorian-diary-listing-hook 'list-hebrew-diary-entries)
(add-hook 'nongregorian-diary-marking-hook 'mark-hebrew-diary-entries)
```

If you want Islamic-date entries, do this:

```
(add-hook 'nongregorian-diary-listing-hook 'list-islamic-diary-entries)
(add-hook 'nongregorian-diary-marking-hook 'mark-islamic-diary-entries)
```

Hebrew- and Islamic-date diary entries have the same formats as Gregorian-date diary entries, except that '`H`' precedes a Hebrew date and '`I`' precedes an Islamic date. Moreover, because the Hebrew and Islamic month names are not uniquely specified by the first three letters, you may not abbreviate them. For example, a diary entry for the Hebrew date Heshvan 25 could look like this:

```
HHeshvan 25 Happy Hebrew birthday!
```

and would appear in the diary for any date that corresponds to Heshvan 25 on the Hebrew calendar. And here is Islamic-date diary entry that matches Dhu al-Qada 25:

```
IDhu al-Qada 25 Happy Islamic birthday!
```

and would appear in the diary for any date that corresponds to Dhu al-Qada 25 on the Islamic calendar.

As with Gregorian-date diary entries, Hebrew- and Islamic-date entries are nonmarking if they are preceded with an ampersand ('`&`').

Here is a table of commands used in the calendar to create diary entries that match the selected date and other dates that are similar in the Hebrew or Islamic calendar:

i h d Add a diary entry for the Hebrew date corresponding to the selected date (`insert-hebrew-diary-entry`).

i h m Add a diary entry for the day of the Hebrew month corresponding to the selected date (`insert-monthly-hebrew-diary-entry`). This diary entry matches any date that has the same Hebrew day-within-month as the selected date.

i h y Add a diary entry for the day of the Hebrew year corresponding to the selected date (`insert-yearly-hebrew-diary-entry`). This diary entry matches any date which has the same Hebrew month and day-within-month as the selected date.

i i d Add a diary entry for the Islamic date corresponding to the selected date (`insert-islamic-diary-entry`).

i i m Add a diary entry for the day of the Islamic month corresponding to the selected date (`insert-monthly-islamic-diary-entry`).

i i y Add a diary entry for the day of the Islamic year corresponding to the selected date (`insert-yearly-islamic-diary-entry`).

These commands work much like the corresponding commands for ordinary diary entries: they apply to the date that point is on in the calendar window, and what they do is insert just the date portion of a diary entry at the end of your diary file. You must then insert the rest of the diary entry.

27.5.8.8 Fancy Diary Display

Diary display works by preparing the diary buffer and then running the hook `diary-display-hook`. The default value of this hook (`simple-diary-display`) hides the irrelevant diary entries and then displays the buffer. However, if you specify the hook as follows,

```
(add-hook 'diary-display-hook 'fancy-diary-display)
```

this enables fancy diary display. It displays diary entries and holidays by copying them into a special buffer that exists only for the sake of display. Copying to a separate buffer provides an opportunity to change the displayed text to make it prettier—for example, to sort the entries by the dates they apply to.

As with simple diary display, you can print a hard copy of the buffer with `print-diary-entries`. To print a hard copy of a day-by-day diary for a week by positioning point on Sunday of that week, type *7 d* and then do *M-x print-diary-entries*. As usual, the inclusion of the holidays slows down the display slightly; you can speed things up by setting the variable `holidays-in-diary-buffer` to `nil`.

Ordinarily, the fancy diary buffer does not show days for which there are no diary entries, even if that day is a holiday. If you want such days to be shown in the fancy diary buffer, set the variable `diary-list-include-blanks` to `t`.

If you use the fancy diary display, you can use the normal hook `list-diary-entries-hook` to sort each day's diary entries by their time of day. Add this line to your '`.emacs`' file:

```
(add-hook 'list-diary-entries-hook 'sort-diary-entries t)
```

For each day, this sorts diary entries that begin with a recognizable time of day according to their times. Diary entries without times come first within each day.

27.5.8.9 Included Diary Files

Fancy diary display also has the ability to process included diary files. This permits a group of people to share a diary file for events that apply to all of them. Lines in the diary file of this form:

```
#include "filename"
```

includes the diary entries from the file *filename* in the fancy diary buffer. The include mechanism is recursive, so that included files can include other files, and so on; you must be careful not to have a cycle of inclusions, of course. Here is how to enable the include facility:

```
(add-hook 'list-diary-entries-hook 'include-other-diary-files)
(add-hook 'mark-diary-entries-hook 'mark-included-diary-files)
```

The include mechanism works only with the fancy diary display, because ordinary diary display shows the entries directly from your diary file.

27.5.8.10 Sexp Entries and the Fancy Diary Display

Sexp diary entries allow you to do more than just have complicated conditions under which a diary entry applies. If you use the fancy diary display, sexp entries can generate the text of the entry depending on the date itself. For example, an anniversary diary entry can insert the number of years since the anniversary date into the text of the diary entry. Thus the '`%d`' in this dairy entry:

```
%%(diary-anniversary 10 31 1948) Arthur's birthday (%d years old)
```

gets replaced by the age, so on October 31, 1990 the entry appears in the fancy diary buffer like this:

```
Arthur's birthday (42 years old)
```

If the diary file instead contains this entry:

```
%%(diary-anniversary 10 31 1948) Arthur's %d%s birthday
```

the entry in the fancy diary buffer for October 31, 1990 appears like this:

```
Arthur's 42nd birthday
```

Similarly, cyclic diary entries can interpolate the number of repetitions that have occurred:

```
%%(diary-cyclic 50 1 1 1990) Renew medication (%d%s time)
```

looks like this:

```
Renew medication (5th time)
```

in the fancy diary display on September 8, 1990.

The generality of sexp diary entries lets you specify any diary entry that you can describe algorithmically. A sexp diary entry contains an expression that computes whether the entry applies to any given date. If its value is non-`nil`, the entry applies to that date; otherwise, it does not. The expression can use the variable `date` to find the date being considered; its value is a list (*month day year*) that refers to the Gregorian calendar.

Suppose you get paid on the 21st of the month if it is a weekday, and on the Friday before if the 21st is on a weekend. Here is how to write a sexp diary entry that matches those dates:

```
&%%(let ((dayname (calendar-day-of-week date))
         (day (car (cdr date))))
      (or (and (= day 21) (memq dayname '(1 2 3 4 5)))
          (and (memq day '(19 20)) (= dayname 5)))
         ) Pay check deposited
```

applies to just those dates. This example illustrates how the sexp can depend on the variable `date`; this variable is a list (*month day year*) that gives the Gregorian date for which the diary entries are being found. If the value of the expression is `t`, the entry applies to that date. If the expression evaluates to `nil`, the entry does *not* apply to that date.

The following sexp diary entries take advantage of the ability (in the fancy diary display) to concoct diary entries whose text varies based on the date:

`%%(diary-sunrise-sunset)`
: Make a diary entry for the local times of today's sunrise and sunset.

`%%(diary-phases-of-moon)`
: Make a diary entry for the phases (quarters) of the moon.

`%%(diary-day-of-year)`
: Make a diary entry with today's day number in the current year and the number of days remaining in the current year.

`%%(diary-iso-date)`
: Make a diary entry with today's equivalent ISO commercial date.

`%%(diary-julian-date)`
: Make a diary entry with today's equivalent date on the Julian calendar.

`%%(diary-astro-day-number)`
: Make a diary entry with today's equivalent astronomical (Julian) day number.

`%%(diary-hebrew-date)`
: Make a diary entry with today's equivalent date on the Hebrew calendar.

`%%(diary-islamic-date)`
: Make a diary entry with today's equivalent date on the Islamic calendar.

`%%(diary-french-date)`
: Make a diary entry with today's equivalent date on the French Revolutionary calendar.

`%%(diary-mayan-date)`
: Make a diary entry with today's equivalent date on the Mayan calendar.

Thus including the diary entry

```
&%%(diary-hebrew-date)
```

causes every day's diary display to contain the equivalent date on the Hebrew calendar, if you are using the fancy diary display. (With simple diary display, the line '`&%%(diary-hebrew-date)`' appears in the diary for any date, but does nothing particularly useful.)

These functions can be used to construct sexp diary entries based on the Hebrew calendar in certain standard ways:

`%%(diary-rosh-hodesh)`
: Make a diary entry that tells the occurrence and ritual announcement of each new Hebrew month.

`%%(diary-parasha)`
: Make a Saturday diary entry that tells the weekly synagogue scripture reading.

`%%(diary-sabbath-candles)`
: Make a Friday diary entry that tells the *local time* of Sabbath candle lighting.

`%%(diary-omer)`
: Make a diary entry that gives the omer count, when appropriate.

`%%(diary-yahrzeit` *month day year*`)` *name*
: Make a diary entry marking the anniversary of a date of death. The date is the *Gregorian* (civil) date of death. The diary entry appears on the proper Hebrew calendar anniversary and on the day before. (In the European style, the order of the parameters is changed to *day*, *month*, *year*.)

27.5.8.11 Customizing Appointment Reminders

You can specify exactly how Emacs reminds you of an appointment, and how far in advance it begins doing so, by setting these variables:

`appt-message-warning-time`
: The time in minutes before an appointment that the reminder begins. The default is 10 minutes.

`appt-audible`
: If this is `t` (the default), Emacs rings the terminal bell for appointment reminders.

`appt-visible`
: If this is `t` (the default), Emacs displays the appointment message in echo area.

`appt-display-mode-line`
: If this is `t` (the default), Emacs displays the number of minutes to the appointment on the mode line.

`appt-msg-window`
: If this is `t` (the default), Emacs displays the appointment message in another window.

`appt-display-duration`
: The number of seconds an appointment message is displayed. The default is 5 seconds.

28 Miscellaneous Commands

This chapter contains several brief topics that do not fit anywhere else.

28.1 Sorting Text

XEmacs provides several commands for sorting text in a 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 using the order determined by the sort keys. The records are ordered so that their keys are in alphabetical order, or, for numerical sorting, in numerical order. In alphabetical sorting, all upper-case letters 'A' through 'Z' come before lower-case 'a', in accordance with the ASCII character sequence.

The sort commands differ in how they divide the text into sort records and in which part of each record they use 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 in 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 in 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 in descending order.

M-x sort-fields
: Divide the region into lines and sort by comparing the contents of one field in each line. Fields are defined as separated by whitespace, so the first run of consecutive non-whitespace characters in a line constitutes field 1, the second such run constitutes field 2, etc.

: You specify which field to sort by with a numeric argument: 1 to sort by field 1, etc. A negative argument means sort in descending order. Thus, minus 2 means sort by field 2 in reverse-alphabetical order.

M-x sort-numeric-fields
: Like *M-x sort-fields*, except the specified field is converted to a number for each line and the numbers are compared. '10' comes before '2' when considered as text, but after it when considered as a number.

M-x sort-columns
: Like *M-x sort-fields*, except that the text within each line used for comparison comes from a fixed range of columns. An explanation is given below.

For example, if the buffer contains:

```
On systems where clash detection (locking of files being edited) is
implemented, XEmacs also checks the first time you modify a buffer
```

```
whether the file has changed on disk since it was last visited or
saved.  If it has, you are asked to confirm that you want to change
the buffer.
```

then if you apply *M-x sort-lines* to the entire buffer you get:

```
On systems where clash detection (locking of files being edited) is
implemented, XEmacs also checks the first time you modify a buffer
saved.  If it has, you are asked to confirm that you want to change
the buffer.
whether the file has changed on disk since it was last visited or
```

where the upper case 'O' comes before all lower case letters. If you apply instead *C-u 2 M-x sort-fields* you get:

```
saved.  If it has, you are asked to confirm that you want to change
implemented, XEmacs also checks the first time you modify a buffer
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 '`If`', '`XEmacs`', '`buffer`', '`systems`', and '`the`'.

M-x sort-columns requires more explanation. You specify the columns by putting point at one of the columns and the mark at the other column. Because this means you cannot put point or the mark at the beginning of the first line to sort, this command uses an unusual definition of 'region': all of the line point is in is considered part of the region, and so is all of the line the mark is in.

For example, to sort a table by information found in columns 10 to 15, you could put the mark on column 10 in the first line of the table, and point on column 15 in the last line of the table, and then use this command. Or you could put the mark on column 15 in the first line and point on column 10 in the last line.

This can be thought of as sorting the rectangle specified by point and the mark, except that the text on each line to the left or right of the rectangle moves along with the text inside the rectangle. See Section 10.5 [Rectangles], page 77.

28.2 Running Shell Commands from XEmacs

XEmacs has commands for passing single command lines to inferior shell processes; it can also run a shell interactively with input and output to an XEmacs 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 XEmacs buffer. You can then give commands interactively.

M-x term Run a subshell with input and output through an XEmacs buffer. You can then give commands interactively. Full terminal emulation is available.

28.2.1 Single Shell Commands

M-! (`shell-command`) reads a line of text using the minibuffer and creates an inferior shell to execute the line as a command. Standard input from the command comes from the null device. If the shell command produces any output, the output goes to an XEmacs buffer named '`*Shell Command Output*`', which is displayed in another window but not selected. A numeric argument, as in *M-1 M-!*, directs this command to insert any output into the current buffer. In that case, point is left before the output and the mark is set after the output.

M-| (`shell-command-on-region`) is like *M-!* but passes the contents of the region as input to the shell command, instead of no input. If a numeric argument is used to direct output to 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 you start XEmacs. 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 `PATH` environment variable when you start XEmacs. You can override either or both of these default initializations in your '`.emacs`' file.

When you use *M-!* and *M-|*, XEmacs has to wait until the shell command completes. You can quit with *C-g*; that terminates the shell command.

28.2.2 Interactive Inferior Shell

To run a subshell interactively with its typescript in an XEmacs buffer, use *M-x shell*. This creates (or reuses) a buffer named '`*shell*`' and runs a subshell with input coming from and output going to that buffer. That is to say, any "terminal output" from the subshell will go into the buffer, advancing point, and any "terminal input" for the subshell comes from text in the buffer. To give input to the subshell, go to the end of the buffer and type the input, terminated by ⟨RET⟩.

XEmacs 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 XEmacs has time to process it; this happens whenever XEmacs is waiting for keyboard input or for time to elapse.

To get multiple subshells, change the name of buffer '`*shell*`' to something different by using *M-x rename-buffer*. The next use of *M-x shell* creates a new buffer '`*shell*`' with its own subshell. By renaming this buffer as well you can create a third one, and so on. All the subshells run independently and in parallel.

The file name used to load the subshell is the value of the variable `explicit-shell-file-name`, if that is non-`nil`. Otherwise, the environment variable `ESHELL` is used, or the environment variable `SHELL` if there is no `ESHELL`. If the file name specified is relative, the directories in the list `exec-path` are searched (see Section 28.2.1 [Single Shell], page 249).

As soon as the subshell is started, it is sent as input the contents of the file '~/.emacs_*shellname*', if that file exists, where *shellname* is the name of the file that the shell was loaded from. For example, if you use csh, the file sent to it is '~/.emacs_csh'.

cd, pushd, and popd commands given to the inferior shell are watched by XEmacs 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 XEmacs 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'.

M-x shell-resync-dirs queries the shell and resynchronizes XEmacs' idea of what the current directory stack is. *M-x shell-dirtrack-toggle* turns directory tracking on and off.

XEmacs keeps a history of the most recent commands you have typed in the '*shell*' buffer. If you are at the beginning of a shell command line and type ⟨M-p⟩, the previous shell input is inserted into the buffer before point. Immediately typing ⟨M-p⟩ again deletes that input and inserts the one before it. By repeating ⟨M-p⟩ you can move backward through your commands until you find one you want to repeat. You may then edit the command before typing ⟨RET⟩ if you wish. ⟨M-n⟩ moves forward through the command history, in case you moved backward past the one you wanted while using ⟨M-p⟩. If you type the first few characters of a previous command and then type ⟨M-p⟩, the most recent shell input starting with those characters is inserted. This can be very convenient when you are repeating a sequence of shell commands. The variable input-ring-size controls how many commands are saved in your input history. The default is 30.

28.2.3 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 XEmacs, except that you must type *C-c* first. Here is a list of the special key bindings of Shell mode:

⟨RET⟩ At end of buffer send line as input; otherwise, copy current line to end of buffer and send it (send-shell-input). When a line is copied, any text at the beginning of the line that matches the variable shell-prompt-pattern is left out; this variable's value should be a regexp string that matches the prompts that you use in your subshell.

C-c C-d Send end-of-file as input, probably causing the shell or its current subjob to finish (shell-send-eof).

C-d If point is not at the end of the buffer, delete the next character just like most other modes. If point is at the end of the buffer, send end-of-file as input, instead of generating an error as in other modes (comint-delchar-or-maybe-eof).

C-c C-u Kill all text that has yet to be sent as input (kill-shell-input).

C-c C-w Kill a word before point (backward-kill-word).

C-c C-c Interrupt the shell or its current subjob if any (interrupt-shell-subjob).

C-c C-z Stop the shell or its current subjob if any (stop-shell-subjob).

C-c C- Send quit signal to the shell or its current subjob if any (quit-shell-subjob).

C-c C-o Delete last batch of output from shell (`kill-output-from-shell`).

C-c C-r Scroll top of last batch of output to top of window (`show-output-from-shell`).

C-c C-y Copy the previous bunch of shell input and insert it into the buffer before point (`copy-last-shell-input`). No final newline is inserted, and the input copied is not resubmitted until you type ⟨RET⟩.

M-p Move backward through the input history. Search for a matching command if you have typed the beginning of a command (`comint-previous-input`).

M-n Move forward through the input history. Useful when you are using ⟨M-p⟩ quickly and go past the desired command (`comint-next-input`).

⟨TAB⟩ Complete the file name preceding point (`comint-dynamic-complete`).

28.2.4 Interactive Inferior Shell with Terminal Emulator

To run a subshell in a terminal emulator, putting its typescript in an XEmacs buffer, use *M-x term*. This creates (or reuses) a buffer named '`*term*`' and runs a subshell with input coming from your keyboard and output going to that buffer.

All the normal keys that you type are sent without any interpretation by XEmacs directly to the subshell, as "terminal input." Any "echo" of your input is the responsibility of the subshell. (The exception is the terminal escape character, which by default is *C-c*. see Section 28.2.5 [Term Mode], page 252.) Any "terminal output" from the subshell goes into the buffer, advancing point.

Some programs (such as XEmacs itself) need to control the appearance on the terminal screen in detail. They do this by sending special control codes. The exact control codes needed vary from terminal to terminal, but nowadays most terminals and terminal emulators (including xterm) understand the so-called "ANSI escape sequences" (first popularized by the Digital's VT100 family of terminal). The term mode also understands these escape sequences, and for each control code does the appropriate thing to change the buffer so that the appearance of the window will match what it would be on a real terminal. Thus you can actually run XEmacs inside an XEmacs Term window!

XEmacs 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 XEmacs has time to process it; this happens whenever XEmacs is waiting for keyboard input or for time to elapse.

To make multiple terminal emulators, rename the buffer '`*term*`' to something different using *M-x rename-uniquely*, just as with Shell mode.

The file name used to load the subshell is determined the same way as for Shell mode.

Unlike Shell mode, Term mode does not track the current directory by examining your input. Instead, if you use a programmable shell, you can have it tell Term what the current directory is. This is done automatically by bash for version 1.15 and later.

28.2.5 Term Mode

Term uses Term mode, which has two input modes: In line mode, Term basically acts like Shell mode. See Section 28.2.3 [Shell Mode], page 250. In Char mode, each character is sent directly to the inferior subshell, except for the Term escape character, normally *C-c*.

To switch between line and char mode, use these commands:

findex term-char-mode

C-c C-k Switch to line mode. Do nothing if already in line mode.

C-c C-j Switch to char mode. Do nothing if already in char mode.

The following commands are only available in Char mode:

C-c C-c Send a literal C-c to the sub-shell.

C-c C-x A prefix command to conveniently access the global C-x commands. For example, *C-c C-x o* invokes the global binding of *C-x o*, which is normally '`other-window`'.

28.2.6 Paging in the terminal emulator

Term mode has a pager feature. When the pager is enabled, term mode will pause at the end of each screenful.

C-c C-q Toggles the pager feature: Disables the pager if it is enabled, and vice versa. This works in both line and char modes. If the pager enabled, the mode-line contains the word '`page`'.

If the pager is enabled, and Term receives more than a screenful of output since your last input, Term will enter More break mode. This is indicated by '`**MORE**`' in the mode-line. Type a *Space* to display the next screenful of output. Type *?* to see your other options. The interface is similar to the Unix '`more`' program.

28.3 Narrowing

Narrowing means focusing in on some portion of the buffer, making the rest temporarily invisible and inaccessible. Cancelling the narrowing and making the entire buffer once again visible is called *widening*. The amount of narrowing in effect in a buffer at any time is called the buffer's *restriction*.

C-x n n Narrow down to between point and mark (`narrow-to-region`).

C-x n w Widen to make the entire buffer visible again (`widen`).

Narrowing sometimes makes it easier to concentrate on a single subroutine or paragraph by eliminating clutter. It can also be used to restrict the range of operation of a replace command or repeating keyboard macro. The word '`Narrow`' appears in the mode line whenever narrowing is in effect. When you have narrowed to a part of the buffer, that part appears to be all there is. You can't see the rest, can't move into it (motion commands won't go outside the visible part), and can't change it in any way. However, the invisible text is not gone; if you save the file, it will be saved.

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 visible but all text before the region or after the region is invisible. Point and mark do not change.

Because narrowing can easily confuse users who do not understand it, `narrow-to-region` is normally a disabled command. Attempting to use this command asks for confirmation and gives you the option of enabling it; once you enable the command, confirmation will no longer be required. See Section 29.4.3 [Disabling], page 274.

To undo narrowing, use *C-x n w* (`widen`). This makes all text in the buffer accessible again.

Use the *C-x =* command to get information on what part of the buffer you narrowed down. See Section 4.8 [Position Info], page 45.

28.4 Hardcopy Output

The XEmacs commands for making hardcopy derive their names from the Unix commands '`print`' and '`lpr`'.

M-x print-buffer
: Print hardcopy of current buffer using Unix command '`print`' ('`lpr -p`'). This command adds page headings containing the file name and page number.

M-x lpr-buffer
: Print hardcopy of current buffer using Unix command '`lpr`'. This command does not add page headings.

M-x print-region
: Like `print-buffer`, but prints only the current region.

M-x lpr-region
: Like `lpr-buffer`, but prints only the current region.

All the hardcopy commands pass extra switches to the `lpr` program based on the value of the variable `lpr-switches`. Its value should be a list of strings, each string a switch starting with '`-`'. For example, the value could be `("-Pfoo")` to print on printer '`foo`'.

28.5 Recursive Editing Levels

A *recursive edit* is a situation in which you are using XEmacs commands to perform arbitrary editing while in the middle of another XEmacs command. For example, when you type *C-r* inside a `query-replace`, you enter a recursive edit in which you can change the current buffer. When you exit from the recursive edit, you go back to the `query-replace`.

Exiting a recursive edit means returning to the unfinished command, which continues execution. For example, exiting the recursive edit requested by *C-r* in `query-replace` causes query replacing to resume. Exiting is done with *C-M-c* (`exit-recursive-edit`).

You can also *abort* a recursive edit. This is like exiting, but also quits the unfinished command immediately. Use the command *C-]* (`abort-recursive-edit`) for this. See Section 30.1 [Quitting], page 291.

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 the square brackets, since XEmacs as a whole, rather than any particular buffer, is in a recursive edit.

It is possible to be in recursive edits within recursive edits. For example, after typing *C-r* in a `query-replace`, you might type a command that entered the debugger. In such a case, two or more sets of square brackets appear in the mode line(s). Exiting the inner recursive edit (here with the debugger *c* command) resumes the query-replace command where it called the debugger. After the end of the query-replace command, you would be able to exit the first recursive edit. Aborting exits only one level of recursive edit; it returns to the command level of the previous recursive edit. You can then abort that one as well.

The command *M-x top-level* aborts all levels of recursive edits, returning immediately to the top level command reader.

The text you edit inside the recursive edit need not be the same text that you were editing at top level. If the command that invokes the recursive edit selects a different buffer first, that is the buffer you will edit recursively. You can switch buffers within the recursive edit in the normal manner (as long as the buffer-switching keys have not been rebound). While you could theoretically do the rest of your editing inside the recursive edit, including visiting files, this could have surprising effects (such as stack overflow) from time to time. It is best if you always exit or abort a recursive edit when you no longer need it.

In general, XEmacs tries to avoid using recursive edits. It is usually preferable to allow users to switch among the possible editing modes in any order they like. With recursive edits, the only way to get to another state is to go "back" to the state that the recursive edit was invoked from.

28.6 Dissociated Press

M-x dissociated-press is a command for scrambling a file of text either word by word or character by character. Starting from a buffer of straight English, it produces extremely amusing output. The input comes from the current XEmacs buffer. Dissociated Press writes its output in a buffer named '`*Dissociation*`', and redisplays that buffer after every couple of lines (approximately) to facilitate reading it.

`dissociated-press` asks every so often whether to continue operating. Answer *n* to stop it. You can also stop at any time by typing *C-g*. The dissociation output remains in the '`*Dissociation*`' buffer for you to copy elsewhere if you wish.

Dissociated Press operates by jumping at random from one point in the buffer to another. In order to produce plausible output rather than gibberish, it insists on a certain amount of overlap between the end of one run of consecutive words or characters and the start of the next. That is, if it has just printed out 'president' and then decides to jump to a different point in the file, it might spot the 'ent' in 'pentagon' and continue from there, producing 'presidentagon'. Long sample texts produce the best results.

A positive argument to *M-x dissociated-press* tells it to operate character by character, and specifies the number of overlap characters. A negative argument tells it to operate word by word and specifies the number of overlap words. In this mode, whole words are treated as the elements to be permuted, rather than characters. No argument is equivalent to an argument of two. For your againformation, the output goes only into the buffer '`*Dissociation*`'. The buffer you start with is not changed.

Dissociated Press produces nearly the same results as a Markov chain based on a frequency table constructed from the sample text. It is, however, an independent, ignoriginal invention. Dissociated Press techniquitously copies several consecutive characters from the sample between random choices, whereas a Markov chain would choose randomly for each word or character. This makes for more plausible sounding results and runs faster.

It is a mustatement that too much use of Dissociated Press can be a developediment to your real work. Sometimes to the point of outragedy. And keep dissociwords out of your documentation, if you want it to be well userenced and properbose. Have fun. Your buggestions are welcome.

28.7 CONX

Besides producing a file of scrambled text with Dissociated Press, you can generate random sentences by using CONX.

M-x conx Generate random sentences in the `*conx*` buffer.

M-x conx-buffer
: Absorb the text in the current buffer into the `conx` database.

M-x conx-init
: Forget the current word-frequency tree.

M-x conx-load
: Load a `conx` database that has been previously saved with `M-x conx-save`.

M-x conx-region
: Absorb the text in the current buffer into the `conx` database.

M-x conx-save
: Save the current `conx` database to a file for future retrieval.

Copy text from a buffer using *M-x conx-buffer* or *M-x conx-region* and then type *M-x conx*. Output is continuously generated until you type ⟨^G⟩. You can save the `conx` database to a file with *M-x conx-save*, which you can retrieve with `M-x conx-load`. To clear the database, use `M-x conx-init`.

28.8 Other Amusements

If you are a little bit bored, you can try *M-x hanoi*. If you are considerably bored, give it a numeric argument. If you are very, very bored, try an argument of 9. Sit back and watch.

When you are frustrated, try the famous Eliza program. Just do *M-x doctor*. End each input by typing *RET* twice.

When you are feeling strange, type *M-x yow*.

28.9 Emulation

XEmacs can be programmed to emulate (more or less) most other editors. Standard facilities can emulate these:

Viper (a vi emulator)

: In XEmacs, Viper is the preferred emulation of vi within XEmacs. Viper is designed to allow you to take advantage of the best features of XEmacs while still doing your basic editing in a familiar, vi-like fashion. Viper provides various different levels of vi emulation, from a quite complete emulation that allows almost no access to native XEmacs commands, to an "expert" mode that combines the most useful vi commands with the most useful XEmacs commands.

  To start Viper, put the command

  ```
  (viper-mode)
  ```

  in your '.emacs' file.

  Viper comes with a separate manual that is provided standard with the XEmacs distribution.

EDT (DEC VMS editor)

: Turn on EDT emulation with *M-x edt-emulation-on*. *M-x edt-emulation-off* restores normal Emacs command bindings.

  Most of the EDT emulation commands are keypad keys, and most standard Emacs key bindings are still available. The EDT emulation rebindings are done in the global keymap, so there is no problem switching buffers or major modes while in EDT emulation.

Gosling Emacs

: Turn on emulation of Gosling Emacs (aka Unipress Emacs) with *M-x set-gosmacs-bindings*. This redefines many keys, mostly on the *C-x* and *ESC* prefixes, to work as they do in Gosmacs. *M-x set-gnu-bindings* returns to normal XEmacs by rebinding the same keys to the definitions they had at the time *M-x set-gosmacs-bindings* was done.

  It is also possible to run Mocklisp code written for Gosling Emacs. See Section 23.3.3 [Mocklisp], page 193.

29 Customization

This chapter talks about various topics relevant to adapting the behavior of Emacs in minor ways.

All kinds of customization affect only the particular Emacs job that you do them in. They are completely lost when you kill the Emacs job, and have no effect on other Emacs jobs you may run at the same time or later. The only way an Emacs job can affect anything outside of it is by writing a file; in particular, the only way to make a customization 'permanent' is to put something in your '`.emacs`' file or other appropriate file to do the customization in each session. See Section 29.6 [Init File], page 277.

29.1 Minor Modes

Minor modes are options which you can use or not. For example, Auto Fill mode is a minor mode in which ⟨SPC⟩ breaks lines between words as you type. All the minor modes are independent of each other and of the selected major mode. Most minor modes inform you in the mode line when they are on; for example, '`Fill`' in the mode line means that Auto Fill mode is on.

Append `-mode` to the name of a minor mode to get the name of a command function that turns the mode on or off. Thus, the command to enable or disable Auto Fill mode is called *`M-x auto-fill-mode`*. These commands are usually invoked with *`M-x`*, but you can bind keys to them if you wish. With no argument, the function turns the mode on if it was off and off if it was on. This is known as *toggling*. A positive argument always turns the mode on, and an explicit zero argument or a negative argument always turns it off.

Auto Fill mode allows you to enter filled text without breaking lines explicitly. Emacs inserts newlines as necessary to prevent lines from becoming too long. See Section 21.6 [Filling], page 159.

Overwrite mode causes ordinary printing characters to replace existing text instead of moving it to the right. For example, if point is in front of the '`B`' in '`FOOBAR`', and you type a *`G`* in Overwrite mode, it changes to '`FOOGAR`', instead of '`FOOGBAR`'.

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 203, for full information.

29.2 Variables

A *variable* is a Lisp symbol which has a value. Variable names can contain any characters, but by convention they are words separated by hyphens. A variable can also 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 has to be a string or 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 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 339).

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 21.6 [Filling], page 159).

29.2.1 Examining and Setting Variables

C-h v
M-x describe-variable
Print the value and documentation of a variable.

M-x set-variable
Change the value of a variable.

To examine the value of a single variable, use *C-h v* (`describe-variable`), which reads a variable name using the minibuffer, with completion. It prints both the value and the documentation of the variable.

```
C-h v fill-column <RET>
```

prints something like:

```
fill-column's value is 75

Documentation:
*Column beyond which automatic line-wrapping should happen.
Automatically becomes local when set in any fashion.
```

The star at the beginning of the documentation indicates that this variable is an option. *C-h v* is not restricted to options; it allows any variable name.

If you know which option you want to set, you can use *M-x set-variable* to set it. This prompts for the variable name in the minibuffer (with completion), and then prompts for 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, as if you had executed the Lisp expression `(setq fill-column 75)`.

Setting variables in this way, like all means of customizing Emacs except where explicitly stated, affects only the current Emacs session.

29.2.2 Easy Customization Interface

A convenient way to find the user option variables that you want to change, and then change them, is with `M-x customize`. This command creates a *customization buffer* with which you can browse through the Emacs user options in a logically organized structure, then edit and set their values. You can also use the customization buffer to save settings permanently. (Not all Emacs user options are included in this structure as of yet, but we are adding the rest.)

29.2.2.1 Customization Groups

For customization purposes, user options are organized into *groups* to help you find them. Groups are collected into bigger groups, all the way up to a master group called `Emacs`.

`M-x customize` creates a customization buffer that shows the top-level `Emacs` group and the second-level groups immediately under it. It looks like this, in part:

```
/- Emacs group: ---------------------------------------------------\
      [State]: visible group members are all at standard settings.
   Customization of the One True Editor.
   See also [Manual].

 [Open] Editing group
Basic text editing facilities.

 [Open] External group
Interfacing to external utilities.

more second-level groups

\- Emacs group end ------------------------------------------------/
```

This says that the buffer displays the contents of the `Emacs` group. The other groups are listed because they are its contents. But they are listed differently, without indentation and dashes, because *their* contents are not included. Each group has a single-line documentation string; the `Emacs` group also has a '`[State]`' line.

Most of the text in the customization buffer is read-only, but it typically includes some *editable fields* that you can edit. There are also *active fields*; this means a field that does something when you *invoke* it. To invoke an active field, either click on it with `Mouse-1`, or move point to it and type ⟨RET⟩.

For example, the phrase '`[Open]`' that appears in a second-level group is an active field. Invoking the '`[Open]`' field for a group opens up a new customization buffer, which shows that group and its contents. This field is a kind of hypertext link to another group.

The `Emacs` group does not include any user options itself, but other groups do. By examining various groups, you will eventually find the options and faces that belong to the feature you are interested in customizing. Then you can use the customization buffer to set them.

You can view the structure of customization groups on a larger scale with `M-x customize-browse`. This command creates a special kind of customization buffer which shows only the names of the groups (and options and faces), and their structure.

In this buffer, you can show the contents of a group by invoking '`[+]`'. When the group contents are visible, this button changes to '`[-]`'; invoking that hides the group contents.

Each group, option or face name in this buffer has an active field which says '`[Group]`', '`[Option]`' or '`[Face]`'. Invoking that active field creates an ordinary customization buffer showing just that group and its contents, just that option, or just that face. This is the way to set values in it.

29.2.2.2 Changing an Option

Here is an example of what a user option looks like in the customization buffer:

```
Kill Ring Max: [Hide] 30
   [State]: this option is unchanged from its standard setting.
Maximum length of kill ring before oldest elements are thrown away.
```

The text following '`[Hide]`', '`30`' in this case, indicates the current value of the option. If you see '`[Show]`' instead of '`[Hide]`', it means that the value is hidden; the customization buffer initially hides values that take up several lines. Invoke '`[Show]`' to show the value.

The line after the option name indicates the *customization state* of the option: in the example above, it says you have not changed the option yet. The word '`[State]`' at the beginning of this line is active; you can get a menu of various operations by invoking it with `Mouse-1` or ⟨RET⟩. These operations are essential for customizing the variable.

The line after the '`[State]`' line displays the beginning of the option's documentation string. If there are more lines of documentation, this line ends with '`[More]`'; invoke this to show the full documentation string.

To enter a new value for '`Kill Ring Max`', move point to the value and edit it textually. For example, you can type `M-d`, then insert another number.

When you begin to alter the text, you will see the '`[State]`' line change to say that you have edited the value:

```
[State]: you have edited the value as text, but not set the option.
```

Editing the value does not actually set the option variable. To do that, you must *set* the option. To do this, invoke the word '`[State]`' and choose '`Set for Current Session`'.

The state of the option changes visibly when you set it:

```
[State]: you have set this option, but not saved it for future sessions.
```

You don't have to worry about specifying a value that is not valid; setting the option checks for validity and will not really install an unacceptable value.

While editing a value or field that is a file name, directory name, command name, or anything else for which completion is defined, you can type *M-*⟨TAB⟩ (`widget-complete`) to do completion.

Some options have a small fixed set of possible legitimate values. These options don't let you edit the value textually. Instead, an active field '`[Value Menu]`' appears before the value; invoke this field to edit the value. For a boolean "on or off" value, the active field says '`[Toggle]`', and it changes to the other value. '`[Value Menu]`' and '`[Toggle]`' edit the buffer; the changes take effect when you use the '`Set for Current Session`' operation.

Some options have values with complex structure. For example, the value of `load-path` is a list of directories. Here is how it appears in the customization buffer:

```
Load Path:
[INS] [DEL] [Current dir?]: /usr/local/share/emacs/19.34.94/site-lisp
[INS] [DEL] [Current dir?]: /usr/local/share/emacs/site-lisp
[INS] [DEL] [Current dir?]: /usr/local/share/emacs/19.34.94/leim
[INS] [DEL] [Current dir?]: /usr/local/share/emacs/19.34.94/lisp
[INS] [DEL] [Current dir?]: /build/emacs/e19/lisp
[INS] [DEL] [Current dir?]: /build/emacs/e19/lisp/gnus
[INS]
   [State]: this item has been changed outside the customization buffer.
List of directories to search for files to load....
```

Each directory in the list appears on a separate line, and each line has several editable or active fields.

You can edit any of the directory names. To delete a directory from the list, invoke '`[DEL]`' on that line. To insert a new directory in the list, invoke '`[INS]`' at the point where you want to insert it.

You can also invoke '`[Current dir?]`' to switch between including a specific named directory in the path, and including `nil` in the path. (`nil` in a search path means "try the current directory.")

Two special commands, ⟨TAB⟩ and *S-*⟨TAB⟩, are useful for moving through the customization buffer. ⟨TAB⟩ (`widget-forward`) moves forward to the next active or editable field; *S-*⟨TAB⟩ (`widget-backward`) moves backward to the previous active or editable field.

Typing ⟨RET⟩ on an editable field also moves forward, just like ⟨TAB⟩. The reason for this is that people have a tendency to type ⟨RET⟩ when they are finished editing a field. If you have occasion to insert a newline in an editable field, use *C-o* or *C-q C-j*,

Setting the option changes its value in the current Emacs session; *saving* the value changes it for future sessions as well. This works by writing code into your '`~/.emacs`' file so as to set the option variable again each time you start Emacs. To save the option, invoke '`[State]`' and select the '`Save for Future Sessions`' operation.

You can also restore the option to its standard value by invoking '[State]' and selecting the 'Reset' operation. There are actually three reset operations:

'Reset to Current'
: If you have made some modifications and not yet set the option, this restores the text in the customization buffer to match the actual value.

'Reset to Saved'
: This restores the value of the option to the last saved value, and updates the text accordingly.

'Reset to Standard Settings'
: This sets the option to its standard value, and updates the text accordingly. This also eliminates any saved value for the option, so that you will get the standard value in future Emacs sessions.

The state of a group indicates whether anything in that group has been edited, set or saved. You can select 'Set for Current Session', 'Save for Future Sessions' and the various kinds of 'Reset' operation for the group; these operations on the group apply to all options in the group and its subgroups.

Near the top of the customization buffer there are two lines containing several active fields:

```
[Set] [Save] [Reset]  [Done]
```

Invoking '[Done]' buries this customization buffer. Each of the other fields performs an operation—set, save or reset—on each of the items in the buffer that could meaningfully be set, saved or reset.

29.2.2.3 Customizing Faces

In addition to user options, some customization groups also include faces. When you show the contents of a group, both the user options and the faces in the group appear in the customization buffer. Here is an example of how a face looks:

```
Custom Changed Face: (sample)
   [State]: this face is unchanged from its standard setting.
Face used when the customize item has been changed.
Parent groups: [Custom Magic Faces]
Attributes: [ ] Bold: [Toggle]  off (nil)
            [ ] Italic: [Toggle]  off (nil)
            [ ] Underline: [Toggle]  off (nil)
            [ ] Foreground: white       (sample)
            [ ] Background: blue        (sample)
            [ ] Inverse: [Toggle]  off (nil)
            [ ] Stipple:
            [ ] Font Family:
            [ ] Size:
            [ ] Strikethru: off
```

Each face attribute has its own line. The '`[x]`' field before the attribute name indicates whether the attribute is *enabled*; '`X`' means that it is. You can enable or disable the attribute by invoking that field. When the attribute is enabled, you can change the attribute value in the usual ways.

Setting, saving and resetting a face work like the same operations for options (see Section 29.2.2.2 [Changing an Option], page 260).

A face can specify different appearances for different types of display. For example, a face can make text red on a color display, but use a bold font on a monochrome display. To specify multiple appearances for a face, select '`Show Display Types`' in the menu you get from invoking '`[State]`'.

29.2.2.4 Customizing Specific Items

Instead of finding the options you want to change by moving down through the structure of groups, you can specify the particular option, face or group that you want to customize.

`M-x customize-option` ⟨RET⟩ *option* ⟨RET⟩
: Set up a customization buffer with just one option, *option*.

`M-x customize-face` ⟨RET⟩ *face* ⟨RET⟩
: Set up a customization buffer with just one face, *face*.

`M-x customize-group` ⟨RET⟩ *group* ⟨RET⟩
: Set up a customization buffer with just one group, *group*.

`M-x customize-apropos` ⟨RET⟩ *regexp* ⟨RET⟩
: Set up a customization buffer with all the options, faces and groups that match *regexp*.

`M-x customize-saved`
: Set up a customization buffer containing all options and faces that you have saved with customization buffers.

`M-x customize-customized`
: Set up a customization buffer containing all options and faces that you have customized but not saved.

If you want to alter a particular user option variable with the customization buffer, and you know its name, you can use the command *`M-x customize-option`* and specify the option name. This sets up the customization buffer with just one option—the one that you asked for. Editing, setting and saving the value work as described above, but only for the specified option.

Likewise, you can modify a specific face, chosen by name, using *`M-x customize-face`*.

You can also set up the customization buffer with a specific group, using *`M-x customize-group`*. The immediate contents of the chosen group, including option variables, faces, and other groups, all appear as well. However, these subgroups' own contents start out hidden. You can show their contents in the usual way, by invoking '`[Show]`'.

To control more precisely what to customize, you can use *`M-x customize-apropos`*. You specify a regular expression as argument; then all options, faces and groups whose names match this regular expression are set up in the customization buffer. If you specify an empty regular expression, this includes *all* groups, options and faces in the customization buffer (but that takes a long time).

If you change option values and then decide the change was a mistake, you can use two special commands to revisit your previous changes. Use *customize-saved* to look at the options and faces that you have saved. Use *M-x customize-customized* to look at the options and faces that you have set but not saved.

29.2.3 Editing Variable Values

M-x list-options
: Display a buffer listing names, values, and documentation of all options.

M-x edit-options
: Change option values by editing a list of options.

M-x list-options displays a list of all Emacs option variables in an Emacs buffer named '`*List Options*`'. Each option is shown with its documentation and its current value. Here is what a portion of it might look like:

```
;; exec-path:
("." "/usr/local/bin" "/usr/ucb" "/bin" "/usr/bin" "/u2/emacs/etc")
*List of directories to search programs to run in subprocesses.
Each element is a string (directory name)
or nil (try the default directory).
;;
;; fill-column:
75
*Column beyond which automatic line-wrapping should happen.
Automatically becomes local when set in any fashion.
;;
```

M-x edit-options goes one step further and immediately selects the '`*List Options*`' buffer; this buffer uses the major mode Options mode, which provides commands that allow you to point at an option and change its value:

s
: Set the variable point is in or near to a new value read using the minibuffer.

x
: Toggle the variable point is in or near: if the value was `nil`, it becomes `t`; otherwise it becomes `nil`.

1
: Set the variable point is in or near to `t`.

0
: Set the variable point is in or near to `nil`.

n
p
: Move to the next or previous variable.

29.2.4 Local Variables

M-x make-local-variable
: Make a variable have a local value in the current buffer.

M-x kill-local-variable
: Make a variable use its global value in the current buffer.

M-x make-variable-buffer-local
: Mark a variable so that setting it will make it local to the buffer that is current at that time.

You can make any variable *local* to a specific Emacs buffer. This means that the variable's value in that buffer is independent of its value in other buffers. A few variables are always local in every buffer. All other Emacs variables have a *global* value which is in effect in all buffers that have not made the variable local.

Major modes always make the variables they set local to the buffer. This is why changing major modes in one buffer has no effect on other buffers.

M-x make-local-variable reads the name of a variable and makes it local to the current buffer. Further changes in this buffer will not affect others, and 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 automatically becomes local when it is set. More precisely, once you have marked a variable in this way, the usual ways of setting the variable will automatically invoke `make-local-variable` first. We call such variables *per-buffer* variables.

Some important variables have been marked per-buffer already. They 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.

Note: the variable `auto-fill-function` was formerly named `auto-fill-hook`.

If you want a variable to cease to be local to the current buffer, call *M-x kill-local-variable* and provide the name of a variable to the prompt. The global value of the variable is again in effect in this buffer. Setting the major mode kills all the local variables of the buffer.

To set the global value of a variable, regardless of whether the variable has a local value in the current buffer, you can use the Lisp function `setq-default`. It works like `setq`. If there is a local value in the current buffer, the local value is not affected by `setq-default`; thus, the new global value may not be visible until you switch to another buffer, as in the case of:

```
(setq-default fill-column 75)
```

`setq-default` is the only way to set the global value of a variable that has been marked with `make-variable-buffer-local`.

Programs can look at a variable's default value with `default-value`. This function takes a symbol as an argument and returns its default value. The argument is evaluated; usually you must quote it explicitly, as in the case of:

```
(default-value 'fill-column)
```

29.2.5 Local Variables in Files

A file can contain a *local variables list*, which specifies the values to use for certain Emacs variables when that file is edited. Visiting the file checks for a local variables list and makes each variable in the list local to the buffer in which the file is visited, with the value specified in the file.

A local variables list goes near the end of the file, in the last page. (It is often best to put it on a page by itself.) The local variables list starts with a line containing the string '`Local Variables:`', and ends with a line containing the string '`End:`'. In between come the variable names and values, one set per line, as '*variable*: *value*'. The *value*s are not evaluated; they are used literally.

The line which starts the local variables list does not have to say just '`Local Variables:`'. If there is other text before '`Local Variables:`', that text is called the *prefix*, and if there is other text after, that is called the *suffix*. If a prefix or suffix are present, each entry in the local variables list should have the prefix before it and the suffix after it. This includes the '`End:`' line. The prefix and suffix are included to disguise the local variables list as a comment so the compiler or text formatter will ignore it. If you do not need to disguise the local variables list as a comment in this way, there is no need to include a prefix or a suffix.

Two "variable" names are special in a local variables list: a value for the variable `mode` sets the major mode, and a value for the variable `eval` is simply evaluated as an expression and the value is ignored. These are not real variables; setting them in any other context does not have the same effect. If `mode` is used in a local variables list, it should be the first entry in the list.

Here is an example of a local variables list:

```
;;; Local Variables: ***
;;; mode:lisp ***
;;; comment-column:0 ***
;;; comment-start: ";;; "  ***
;;; comment-end:"***" ***
;;; End: ***
```

Note that the prefix is '`;;; `' and the suffix is '` ***`'. Note also that comments in the file begin with and end with the same strings. Presumably the file contains code in a language which is enough like Lisp for Lisp mode to be useful but in which comments start and end differently. The prefix and suffix are used in the local variables list to make the list look like several lines of comments when the compiler or interpreter for that language reads the file.

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 is twofold: a stray '`Local Variables:`' not in the last page does not confuse Emacs, and Emacs never needs to search a long file that contains no page markers and has no local variables list.

You may be tempted to turn on Auto Fill mode with a local variable list. That is inappropriate. Whether you use Auto Fill mode or not is a matter of personal 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 29.6 [Init File], page 277). Don't try to use a local variable list that would impose your taste on everyone working with the file.

XEmacs allows you to specify local variables in the first line of a file, in addition to specifying them in the `Local Variables` section at the end of a file.

If the first line of a file contains two occurrences of '`-*-`', XEmacs uses the information between them to determine what the major mode and variable settings should be. For example, these are all legal:

```
;;; -*- mode: emacs-lisp -*-
;;; -*- mode: postscript; version-control: never -*-
;;; -*- tags-file-name: "/foo/bar/TAGS" -*-
```

For historical reasons, the syntax '`-*- modename -*-`' is allowed as well; for example, you can use:

```
;;; -*- emacs-lisp -*-
```

The variable `enable-local-variables` controls the use of local variables lists in files you visit. The value can be `t`, `nil`, or something else. A value of `t` means local variables lists are obeyed; `nil` means they are ignored; anything else means query.

The command `M-x normal-mode` always obeys local variables lists and ignores this variable.

29.3 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 invoke *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 other 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 general or complex. For such things, Lisp must be used.

You define a keyboard macro by executing the commands which are its definition. Put differently, as you are defining a keyboard macro, the definition is being executed for the first time. This way, you see what the effects of your commands are, and don't have to figure them out in your head. When you are finished, the keyboard macro is defined and also has been executed once. You can then execute the same set of commands again by invoking the macro.

29.3.1 Basic Use

To start defining a keyboard macro, type *C-x (* (`start-kbd-macro`). From then on, anything you type continues to be executed, but also becomes 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`'.

You can give *C-x)* 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). If you give *C-x)* an argument of 4, it 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*).

Once you have defined a macro, you can invoke it again with the *C-x e* command (`call-last-kbd-macro`). You can give the command a repeat count numeric argument to execute the macro many times.

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. Repeating the macro will then 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.

29.3.2 Naming and Saving Keyboard Macros

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 29.4.1 [Keymaps], page 270). If you specify a name that has a prior definition other than another keyboard macro, Emacs prints an error message and nothing is changed.

Once a macro has a command name, you can save its definition in a file. You can then use it in another editing session. First visit the file you want to save the definition in. Then use the command:

```
M-x insert-kbd-macro ⟨RET⟩ macroname ⟨RET⟩
```

This inserts some Lisp code that, when executed later, will define the same macro with the same definition it has now. You need not understand Lisp code to do this, because `insert-kbd-macro` writes the Lisp code for you. Then save the file. You can load the file with `load-file` (see Section 23.3 [Lisp Libraries], page 191). If the file you save in is your initialization file '`~/.emacs`' (see Section 29.6 [Init File], page 277), then the macro will be defined each time you run Emacs.

If you give `insert-kbd-macro` a prefix argument, it creates additional Lisp code to record the keys (if any) that you have bound to the keyboard macro, so that the macro is reassigned the same keys when you load the file.

29.3.3 Executing Macros With Variations

You can use *C-x q* (`kbd-macro-query`), to get an effect similar to that of `query-replace`. The macro asks you each time 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 invoke the macro, *C-x q* reads a character from the terminal to decide whether to continue.

The special answers to a *C-x q* query are ⟨SPC⟩, ⟨DEL⟩, *C-d*, *C-l*, and *C-r*. Any other character terminates execution of the keyboard macro and is then read as a command. ⟨SPC⟩ means to continue. ⟨DEL⟩ means to skip the remainder of this repetition of the macro, starting again from the beginning in the next repetition. *C-d* means to skip the remainder of this repetition and cancel further repetition. *C-l* redraws the frame and asks you again for a character to specify what to do. *C-r* enters a recursive editing level, in which you can perform editing that 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 28.5 [Recursive Edit], page 253.

29.4 Customizing Key Bindings

This section deals with the *keymaps* that define the bindings between keys and functions, and shows how you can customize these bindings.

A command is a Lisp function whose definition provides for interactive use. Like every Lisp function, a command has a function name, which is a Lisp symbol whose name usually consists of lower case letters and hyphens.

29.4.1 Keymaps

The bindings between characters and command functions are recorded in data structures called *keymaps*. Emacs has many of these. One, the *global* keymap, defines the meanings of the single-character keys that are defined regardless of major mode. It is the value of the variable `global-map`.

Each major mode has another keymap, its *local keymap*, which contains overriding definitions for the single-character keys that are redefined in that mode. Each buffer records which local keymap is installed for it at any time, and the current buffer's local keymap is the only one that directly affects command execution. The local keymaps for Lisp mode, C mode, and many other major modes always exist even when not in use. They are the values of the variables `lisp-mode-map`, `c-mode-map`, and so on. For less frequently used major modes, the local keymap is sometimes constructed only when the mode is used for the first time in a session, to save space.

There are local keymaps for the minibuffer, too; they contain various completion and exit commands.

- `minibuffer-local-map` is used for ordinary input (no completion).
- `minibuffer-local-ns-map` is similar, except that ⟨SPC⟩ exits just like ⟨RET⟩. This is used mainly for Mocklisp compatibility.
- `minibuffer-local-completion-map` is for permissive completion.
- `minibuffer-local-must-match-map` is for strict completion and for cautious completion.
- `repeat-complex-command-map` is for use in *C-x* ⟨ESC⟩.
- `isearch-mode-map` contains the bindings of the special keys which are bound in the pseudo-mode entered with *C-s* and *C-r*.

Finally, each prefix key has a keymap which defines the key sequences that start with it. For example, `ctl-x-map` is the keymap used for characters following a *C-x*.

- `ctl-x-map` is the variable name for the map used for characters that follow *C-x*.
- `help-map` is used for characters that follow *C-h*.
- `esc-map` is for characters that follow ⟨ESC⟩. All Meta characters are actually defined by this map.
- `ctl-x-4-map` is for characters that follow *C-x 4*.
- `mode-specific-map` is for characters that follow *C-c*.

The definition of a prefix key is the keymap to use for looking up the following character. Sometimes the definition is actually a Lisp symbol whose function definition is the following character keymap. The effect is the same, but it provides a command name for the prefix key that you can use as a description of what the prefix key is for. Thus the binding of *C-x* is the symbol `Ctl-X-Prefix`, whose function definition is the keymap for *C-x* commands, the value of `ctl-x-map`.

Prefix key definitions can appear in either the global map or a local map. The definitions of *C-c*, *C-x*, *C-h*, and ⟨ESC⟩ as prefix keys appear in the global map, so these prefix keys are always available. Major modes can locally redefine a key as a prefix by putting a prefix key definition for it in the local map.

A mode can also put a prefix definition of a global prefix character such as *C-x* into its local map. This is how major modes override the definitions of certain keys that start with *C-x*. This case is special, because the local definition does not entirely replace the global one. When both the global and local definitions of a key are other keymaps, the next character is looked up in both keymaps, with the local definition overriding the global one. The character after the *C-x* is looked up in both the major mode's own keymap for redefined *C-x* commands and in `ctl-x-map`. If the major mode's own keymap for *C-x* commands contains `nil`, the definition from the global keymap for *C-x* commands is used.

29.4.2 Changing Key Bindings

You can redefine an Emacs key by changing 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.

29.4.2.1 Changing Key Bindings Interactively

M-x global-set-key ⟨RET⟩ *key cmd* ⟨RET⟩
: Defines *key* globally to run *cmd*.

M-x local-set-key ⟨RET⟩ *keys cmd* ⟨RET⟩
: Defines *key* locally (in the major mode now in effect) to run *cmd*.

M-x local-unset-key ⟨RET⟩ *keys* ⟨RET⟩
: Removes the local binding of *key*.

cmd is a symbol naming an interactively-callable function.

When called interactively, *key* is the next complete key sequence that you type. When called as a function, *key* is a string, a vector of events, or a vector of key-description lists as described in the `define-key` function description. The binding goes in the current buffer's local map, which is shared with other buffers in the same major mode.

The following example:

```
M-x global-set-key ⟨RET⟩ C-f next-line ⟨RET⟩
```

redefines *C-f* to move down a line. The fact that *cmd* is read second makes it serve as a kind of confirmation for *key*.

These functions offer no way to specify a particular prefix keymap as the one to redefine in, but that is not necessary, as you can include prefixes in *key*. *key* is read by reading characters one by one until they amount to a complete key (that is, not a prefix key). Thus, if you type *C-f* for *key*, Emacs enters the minibuffer immediately to read *cmd*. But if you type *C-x*, another character is read; if that character is *4*, another character is read, and so on. For example,

```
M-x global-set-key ⟨RET⟩ C-x 4 $ spell-other-window ⟨RET⟩
```

redefines *C-x 4 $* to run the (fictitious) command `spell-other-window`.

The most general way to modify a keymap is the function `define-key`, used in Lisp code (such as your '`.emacs`' file). `define-key` takes three arguments: the keymap, the key to modify in it, and the new definition. See Section 29.6 [Init File], page 277, for an example. `substitute-key-definition` is used similarly; it takes three arguments, an old definition, a new definition, and a keymap, and redefines in that keymap all keys that were previously defined with the old definition to have the new definition instead.

29.4.2.2 Changing Key Bindings Programmatically

You can use the functions `global-set-key` and `define-key` to rebind keys under program control.

`(global-set-key` *keys cmd*`)`
: Defines *keys* globally to run *cmd*.

`(define-key` *keymap keys def*`)`
: Defines *keys* to run *def* in the keymap *keymap*.

keymap is a keymap object.

keys is the sequence of keystrokes to bind.

def is anything that can be a key's definition:

- `nil`, meaning key is undefined in this keymap
- A command, that is, a Lisp function suitable for interactive calling
- A string or key sequence vector, which is treated as a keyboard macro
- A keymap to define a prefix key
- A symbol so that when the key is looked up, the symbol stands for its function definition, which should at that time be one of the above, or another symbol whose function definition is used, and so on
- A cons, `(string . defn)`, meaning that *defn* is the definition (*defn* should be a valid definition in its own right)
- A cons, `(keymap . char)`, meaning use the definition of *char* in map *keymap*

For backward compatibility, XEmacs allows you to specify key sequences as strings. However, the preferred method is to use the representations of key sequences as vectors of keystrokes. See Chapter 2 [Keystrokes], page 19, for more information about the rules for constructing key sequences.

Emacs allows you to abbreviate representations for key sequences in most places where there is no ambiguity. Here are some rules for abbreviation:

- The keysym by itself is equivalent to a list of just that keysym, i.e., `f1` is equivalent to `(f1)`.
- A keystroke by itself is equivalent to a vector containing just that keystroke, i.e., `(control a)` is equivalent to `[(control a)]`.
- You can use ASCII codes for keysyms that have them. i.e., `65` is equivalent to `A`. (This is not so much an abbreviation as an alternate representation.)

Here are some examples of programmatically binding keys:

```
;;;  Bind my-command to f1
(global-set-key 'f1 'my-command)

;;;  Bind my-command to Shift-f1
(global-set-key '(shift f1) 'my-command)

;;; Bind my-command to C-c Shift-f1
(global-set-key '[(control c) (shift f1)] 'my-command)

;;; Bind my-command to the middle mouse button.
(global-set-key 'button2 'my-command)

;;; Bind my-command to META CTL Right Mouse Button
;;; in the keymap that is in force when you are running dired.
(define-key dired-mode-map '(meta control button3) 'my-command)
```

29.4.2.3 Using Strings for Changing Key Bindings

For backward compatibility, you can still use strings to represent key sequences. Thus you can use commands like the following:

```
;;; Bind end-of-line to C-f
(global-set-key "\C-f" 'end-of-line)
```

Note, however, that in some cases you may be binding more than one key sequence by using a single command. This situation can arise because in ASCII, `C-i` and TAB have the same representation. Therefore, when Emacs sees:

```
(global-set-key "\C-i" 'end-of-line)
```

it is unclear whether the user intended to bind `C-i` or TAB. The solution XEmacs adopts is to bind both of these key sequences.

After binding a command to two key sequences with a form like:

```
(define-key global-map "\^X\^I" 'command-1)
```

it is possible to redefine only one of those sequences like so:

```
(define-key global-map [(control x) (control i)] 'command-2)
(define-key global-map [(control x) tab] 'command-3)
```

This applies only when running under a window system. If you are talking to Emacs through an ASCII-only channel, you do not get any of these features.

Here is a table of pairs of key sequences that behave in a similar fashion:

```
control h      backspace
control l      clear
control i      tab
control m      return
control j      linefeed
control [      escape
control @      control space
```

29.4.3 Disabling Commands

Disabling a command marks it as requiring confirmation before it can be executed. The purpose of disabling a command is to prevent beginning users from executing it by accident and being confused.

The direct mechanism for disabling a command is to have a non-`nil` `disabled` property on the Lisp symbol for the command. These properties are normally set by the user's '`.emacs`' file with Lisp expressions such as:

```
(put 'delete-region 'disabled t)
```

If the value of the `disabled` property is a string, that string is included in the message printed when the command is used:

```
(put 'delete-region 'disabled
     "Text deleted this way cannot be yanked back!\n")
```

You can disable a command either by editing the '`.emacs`' file directly or with the command *`M-x disable-command`*, which edits the '`.emacs`' file for you. See Section 29.6 [Init File], page 277.

When you attempt to invoke a disabled command interactively in Emacs, a window is displayed containing the command's name, its documentation, and some instructions on what to do next; then Emacs asks for input saying whether to execute the command as requested, enable it and execute, or cancel it. If you decide to enable the command, you are asked whether to do this permanently or just for the current session. Enabling permanently works by automatically editing your '`.emacs`' file. You can use *`M-x enable-command`* at any time to enable any command permanently.

Whether a command is disabled is independent of what key is used to invoke it; it also applies if the command is invoked using *`M-x`*. Disabling a command has no effect on calling it as a function from Lisp programs.

29.5 The Syntax Table

All the Emacs commands which parse words or balance parentheses are controlled by the *syntax table*. The syntax table specifies which characters are opening delimiters, which are parts of words, which are string quotes, and so on. Actually, each major mode has its own syntax table (though sometimes related major modes use the same one) which it installs in each buffer that uses that major mode. The syntax table installed in the current buffer is the one that all commands use, so we call it "the" syntax table. A syntax table is a Lisp object, a vector of length 256 whose elements are numbers.

29.5.1 Information About Each Character

The syntax table entry for a character is a number that encodes six pieces of information:

- The syntactic class of the character, represented as a small integer
- The matching delimiter, for delimiter characters only (the matching delimiter of '(' is ')', and vice versa)
- A flag saying whether the character is the first character of a two-character comment starting sequence
- A flag saying whether the character is the second character of a two-character comment starting sequence
- A flag saying whether the character is the first character of a two-character comment ending sequence
- A flag saying whether the character is the second character of a two-character comment ending sequence

The syntactic classes are stored internally as small integers, but are usually described to or by the user with characters. For example, '(' is used to specify the syntactic class of opening delimiters. Here is a table of syntactic classes, with the characters that specify them.

' ' The class of whitespace characters.

'w' The class of word-constituent characters.

'_' The class of characters that are part of symbol names but not words. This class is represented by '_' because the character '_' has this class in both C and Lisp.

'.' The class of punctuation characters that do not fit into any other special class.

'(' The class of opening delimiters.

')' The class of closing delimiters.

''' The class of expression-adhering characters. These characters are part of a symbol if found within or adjacent to one, and are part of a following expression if immediately preceding one, but are like whitespace if surrounded by whitespace.

'"' The class of string-quote characters. They match each other in pairs, and the characters within the pair all lose their syntactic significance except for the '\' and '/' classes of escape characters, which can be used to include a string-quote inside the string.

'$' The class of self-matching delimiters. This is intended for TeX's '$', which is used both to enter and leave math mode. Thus, a pair of matching '$' characters surround each piece of math mode TeX input. A pair of adjacent '$' characters act like a single one for purposes of matching.

'/' The class of escape characters that always just deny the following character its special syntactic significance. The character after one of these escapes is always treated as alphabetic.

'\' The class of C-style escape characters. In practice, these are treated just like '/'-class characters, because the extra possibilities for C escapes (such as being followed by digits) have no effect on where the containing expression ends.

'<' The class of comment-starting characters. Only single-character comment starters (such as ';' in Lisp mode) are represented this way.

'>' The class of comment-ending characters. Newline has this syntax in Lisp mode.

The characters flagged as part of two-character comment delimiters can have other syntactic functions most of the time. For example, '/' and '*' in C code, when found separately, have nothing to do with comments. The comment-delimiter significance overrides when the pair of characters occur together in the proper order. Only the list and sexp commands use the syntax table to find comments; the commands specifically for comments have other variables that tell them where to find comments. Moreover, the list and sexp commands notice comments only if `parse-sexp-ignore-comments` is non-`nil`. This variable is set to `nil` in modes where comment-terminator sequences are liable to appear where there is no comment, for example, in Lisp mode where the comment terminator is a newline but not every newline ends a comment.

29.5.2 Altering Syntax Information

It is possible to alter a character's syntax table entry by storing a new number in the appropriate element of the syntax table, but it would be hard to determine what number to use. Emacs therefore provides a command that allows you to specify the syntactic properties of a character in a convenient way.

M-x modify-syntax-entry is the command to change a character's syntax. It can be used interactively and is also used by major modes to initialize their own syntax tables. Its first argument is the character to change. The second argument is a string that specifies the new syntax. When called from Lisp code, there is a third, optional argument, which specifies the syntax table in which to make the change. If not supplied, or if this command is called interactively, the third argument defaults to the current buffer's syntax table.

1. The first character in the string specifies the syntactic class. It is one of the characters in the previous table (see Section 29.5.1 [Syntax Entry], page 275).
2. The second character is the matching delimiter. For a character that is not an opening or closing delimiter, this should be a space, and may be omitted if no following characters are needed.
3. The remaining characters are flags. The flag characters allowed are:

   '1' Flag this character as the first of a two-character comment starting sequence.

   '2' Flag this character as the second of a two-character comment starting sequence.

   '3' Flag this character as the first of a two-character comment ending sequence.

   '4' Flag this character as the second of a two-character comment ending sequence.

Use *C-h s* (`describe-syntax`) to display a description of the contents of the current syntax table. The description of each character includes both the string you have to pass to `modify-syntax-entry` to set up that character's current syntax, and some English to explain that string if necessary.

29.6 The Init File, .emacs

When you start Emacs, it normally loads the file '`.emacs`' in your home directory. This file, if it exists, should contain Lisp code. It is called your initialization file or *init file*. Use the command line switches '`-q`' and '`-u`' to tell Emacs whether to load an init file (see Chapter 3 [Entering Emacs], page 33).

When the '`.emacs`' file is read, the variable `init-file-user` says which user's init file it is. The value may be the null string or a string containing a user's name. If the value is a null string, it means that the init file was taken from the user that originally logged in.

In all cases, `(concat "~" init-file-user "/")` evaluates to the directory name of the directory where the '`.emacs`' file was looked for.

At some sites there is a *default init file*, which is the library named '`default.el`', found via the standard search path for libraries. The Emacs distribution contains no such library; your site may create one for local customizations. If this library exists, it is loaded whenever you start Emacs. But your init file, if any, is loaded first; if it sets `inhibit-default-init` non-`nil`, then '`default`' is not loaded.

If you have a large amount of code in your '`.emacs`' file, you should move it into another file named '*something*`.el`', byte-compile it (see Section 23.3 [Lisp Libraries], page 191), and load that file from your '`.emacs`' file using `load`.

29.6.1 Init File Syntax

The '`.emacs`' file contains one or more Lisp function call expressions. Each consists of a function name followed by arguments, all surrounded by parentheses. For example, `(setq fill-column 60)` represents a call to the function `setq` which is used to set the variable `fill-column` (see Section 21.6 [Filling], page 159) 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
: Integers are written in decimal, with an optional initial minus sign.

  If a sequence of digits is followed by a period and another sequence of digits, it is interpreted as a floating point number.

  The number prefixes '`#b`', '`#o`', and '`#x`' are supported to represent numbers in binary, octal, and hexadecimal notation (or radix).

Strings
: Lisp string syntax is the same as C string syntax with a few extra features. Use a double-quote character to begin and end a string constant.

  Newlines and special characters may be present literally in strings. They can also be represented as backslash sequences: '`\n`' for newline, '`\b`' for backspace, '`\r`' for 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.

You can use '\C-' as a prefix for a control character, as in '\C-s' for ASCII Control-S, and '\M-' 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.

29.6.2 Init File Examples

Here are some examples of doing certain commonly desired things with Lisp expressions:

- Make ⟨TAB⟩ in C mode just insert a tab if point is in the middle of a line.

  ```
  (setq c-tab-always-indent nil)
  ```

  Here we have a variable whose value is normally `t` for 'true' and the alternative is `nil` for 'false'.
- Make searches case sensitive by default (in all buffers that do not override this).

  ```
  (setq-default case-fold-search nil)
  ```

  This sets the default value, which is effective in all buffers that do not have local values for the variable. Setting `case-fold-search` with `setq` affects only the current buffer's local value, which is probably not what you want to do in an init file.
- Make Text mode the default mode for new buffers.

  ```
  (setq default-major-mode 'text-mode)
  ```

  Note that `text-mode` is used because it is the command for entering the mode we want. A single-quote is written before it to make a symbol constant; otherwise, `text-mode` would be treated as a variable name.
- Turn on Auto Fill mode automatically in Text mode and related modes.

  ```
  (setq text-mode-hook
    '(lambda () (auto-fill-mode 1)))
  ```

  Here we have a variable whose value should be a Lisp function. The function we supply is a list starting with `lambda`, and a single quote is written in front of it to make it (for the purpose of this `setq`) a list constant rather than an expression. Lisp functions are not explained here; for mode hooks it is enough to know that `(auto-fill-mode 1)` is an expression that will be executed when Text mode is entered. You could replace it with any other expression that you like, or with several expressions in a row.

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

  This is another way to accomplish the same result. `turn-on-auto-fill` is a symbol whose function definition is `(lambda () (auto-fill-mode 1))`.
- Load the installed Lisp library named '`foo`' (actually a file '`foo.elc`' or '`foo.el`' in a standard Emacs directory).

```
(load "foo")
```

When the argument to `load` is a relative pathname, not starting with '`/`' or '`~`', `load` searches the directories in `load-path` (see Section 23.3.1 [Loading], page 191).

- 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)
  ```
- Bind the function key ⟨F1⟩ to a command in C mode. Note that the names of function keys must be lower case.

  ```
  (define-key c-mode-map 'f1 'make-symbolic-link)
  ```
- Bind the shifted version of ⟨F1⟩ to a command.

  ```
  (define-key c-mode-map '(shift f1) 'make-symbolic-link)
  ```
- Redefine all keys which now run `next-line` in Fundamental mode to run `forward-line` instead.

  ```
  (substitute-key-definition 'next-line 'forward-line
                             global-map)
  ```
- Make *C-x C-v* undefined.

  ```
  (global-unset-key "\C-x\C-v")
  ```

  One reason to undefine a key is so that you can make it a prefix. Simply defining *C-x C-v anything* would make *C-x C-v* a prefix, but *C-x C-v* must be freed of any non-prefix definition first.
- Make '`$`' have the syntax of punctuation in Text mode. Note the use of a character constant for '`$`'.

  ```
  (modify-syntax-entry ?\$ "." text-mode-syntax-table)
  ```
- Enable the use of the command `eval-expression` without confirmation.

  ```
  (put 'eval-expression 'disabled nil)
  ```

29.6.3 Terminal-Specific Initialization

Each terminal type can have a Lisp library to be loaded into Emacs when it is run on that type of terminal. For a terminal type named *termtype*, the library is called '`term/`*termtype*' and it is found by searching the directories `load-path` as usual and trying the suffixes '`.elc`' and '`.el`'. Normally it appears in the subdirectory '`term`' of the directory where most Emacs libraries are kept.

The usual purpose of the terminal-specific library is to define the escape sequences used by the terminal's function keys using the library '`keypad.el`'. See the file '`term/vt100.el`' for an example of how this is done.

When the terminal type contains a hyphen, only the part of the name before the first hyphen is significant in choosing the library name. Thus, terminal types '`aaa-48`' and '`aaa-30-rv`' both use the library '`term/aaa`'. The code in the library can use `(getenv "TERM")` to find the full terminal type name.

The library's name is constructed by concatenating the value of the variable `term-file-prefix` and the terminal type. Your '`.emacs`' file can prevent the loading of the terminal-specific library by setting `term-file-prefix` to `nil`.

The value of the variable `term-setup-hook`, if not `nil`, is called as a function of no arguments at the end of Emacs initialization, after both your '`.emacs`' file and any terminal-specific library have been read. You can set the value in the '`.emacs`' file to override part of any of the terminal-specific libraries and to define initializations for terminals that do not have a library.

29.7 Changing the Bell Sound

You can now change how the audible bell sounds using the variable `sound-alist`.

`sound-alist`'s value is an list associating symbols with, among other things, strings of audio-data. When `ding` is called with one of the symbols, the associated sound data is played instead of the standard beep. This only works if you are logged in on the console of a machine with audio hardware. To listen to a sound of the provided type, call the function `play-sound` with the argument *sound*. You can also set the volume of the sound with the optional argument *volume*.

Each element of `sound-alist` is a list describing a sound. The first element of the list is the name of the sound being defined. Subsequent elements of the list are alternating keyword/value pairs:

`sound` A string of raw sound data, or the name of another sound to play. The symbol `t` here means use the default X beep.

`volume` An integer from 0-100, defaulting to `bell-volume`.

`pitch` If using the default X beep, the pitch (Hz) to generate.

`duration` If using the default X beep, the duration (milliseconds).

For compatibility, elements of 'sound-alist' may also be of the form:

```
( sound-name . <sound> )
( sound-name <volume> <sound> )
```

You should probably add things to this list by calling the function `load-sound-file`.

Note that you can only play audio data if running on the console screen of a machine with audio hardware which emacs understands, which at this time means a Sun SparcStation, SGI, or HP9000s700.

Also note that the pitch, duration, and volume options are available everywhere, but most X servers ignore the 'pitch' option.

The variable `bell-volume` should be an integer from 0 to 100, with 100 being loudest, which controls how loud the sounds emacs makes should be. Elements of the `sound-alist` may override this value. This variable applies to the standard X bell sound as well as sound files.

If the symbol `t` is in place of a sound-string, Emacs uses the default X beep. This allows you to define beep-types of different volumes even when not running on the console.

You can add things to this list by calling the function `load-sound-file`, which reads in an audio-file and adds its data to the sound-alist. You can specify the sound with the *sound-name* argument and the file into which the sounds are loaded with the *filename* argument. The optional *volume* argument sets the volume.

`load-sound-file` (*filename sound-name* `&optional` *volume*)

To load and install some sound files as beep-types, use the function `load-default-sounds` (note that this only works if you are on display 0 of a machine with audio hardware).

The following beep-types are used by Emacs itself. Other Lisp packages may use other beep types, but these are the ones that the C kernel of Emacs uses.

`auto-save-error`
: An auto-save does not succeed

`command-error`
: The Emacs command loop catches an error

`undefined-key`
: You type a key that is undefined

`undefined-click`
: You use an undefined mouse-click combination

`no-completion`
: Completion was not possible

`y-or-n-p`
: You type something other than the required `y` or `n`

`yes-or-no-p`
: You type something other than `yes` or `no`

29.8 Faces

XEmacs has objects called extents and faces. An *extent* is a region of text and a *face* is a collection of textual attributes, such as fonts and colors. Every extent is displayed in some face; therefore, changing the properties of a face immediately updates the display of all associated extents. Faces can be frame-local: you can have a region of text that displays with completely different attributes when its buffer is viewed from a different X window.

The display attributes of faces may be specified either in Lisp or through the X resource manager.

29.8.1 Customizing Faces

You can change the face of an extent with the functions in this section. All the functions prompt for a *face* as an argument; use completion for a list of possible values.

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

M-x make-face-bold
: Make the font of the given *face* bold. When called from a program, returns `nil` if this is not possible.

M-x make-face-bold-italic
: Make the font of the given *face* bold italic. When called from a program, returns `nil` if not possible.

M-x make-face-italic
: Make the font of the given *face* italic. When called from a program, returns `nil` if not possible.

M-x make-face-unbold
: Make the font of the given *face* non-bold. When called from a program, returns `nil` if not possible.

M-x make-face-unitalic
: Make the font of the given *face* non-italic. When called from a program, returns `nil` if not possible.

M-x make-face-larger
: Make the font of the given *face* a little larger. When called from a program, returns `nil` if not possible.

M-x make-face-smaller
: Make the font of the given *face* a little smaller. When called from a program, returns `nil` if not possible.

M-x set-face-background
: Change the background color of the given *face*.

M-x set-face-background-pixmap
: Change the background pixmap of the given *face*.

M-x set-face-font
: Change the font of the given *face*.

M-x set-face-foreground
: Change the foreground color of the given *face*.

M-x set-face-underline-p
: Change whether the given *face* is underlined.

You can exchange the foreground and background color of the selected *face* with the function `invert-face`. If the face does not specify both foreground and background, then its foreground and background are set to the background and foreground of the default face. When calling this from a program, you can supply the optional argument *frame* to specify which frame is affected; otherwise, all frames are affected.

You can set the background color of the specified *face* with the function `set-face-background`. The argument `color` should be a string, the name of a color. When called from a program, if the optional *frame* argument is provided, the face is changed only in that frame; otherwise, it is changed in all frames.

You can set the background pixmap of the specified *face* with the function `set-face-background-pixmap`. The pixmap argument *name* should be a string, the name of a file of pixmap data. The directories listed in the `x-bitmap-file-path` variable are searched. The bitmap may also be a list of the form `(`*width height data*`)`, where *width* and *height* are the size in pixels, and *data* is a string containing the raw bits of the bitmap. If the optional *frame* argument is provided, the face is changed only in that frame; otherwise, it is changed in all frames.

The variable `x-bitmap-file-path` takes as a value a list of the directories in which X bitmap files may be found. If the value is `nil`, the list is initialized from the `*bitmapFilePath` resource.

If the environment variable **XBMLANGPATH** is set, then it is consulted before the `x-bitmap-file-path` variable.

You can set the font of the specified *face* with the function `set-face-font`. The *font* argument should be a string, the name of a font. When called from a program, if the optional *frame* argument is provided, the face is changed only in that frame; otherwise, it is changed in all frames.

You can set the foreground color of the specified *face* with the function `set-face-foreground`. The argument *color* should be a string, the name of a color. If the optional *frame* argument is provided, the face is changed only in that frame; otherwise, it is changed in all frames.

You can set underline the specified *face* with the function `set-face-underline-p`. The argument *underline-p* can be used to make underlining an attribute of the face or not. If the optional *frame* argument is provided, the face is changed only in that frame; otherwise, it is changed in all frames.

29.9 X Resources

Historically, XEmacs has used the X resource application class '`Emacs`' for its resources. Unfortunately, GNU Emacs uses the same application class, and resources are not compatible between the two Emacsen. This sharing of the application class often leads to trouble if you want to run both variants.

Starting with XEmacs 21, XEmacs uses the class '`XEmacs`' if it finds any XEmacs resources in the resource database when the X connection is initialized. Otherwise, it will use the class '`Emacs`' for backwards compatability. The variable *x-emacs-application-class* may be consulted to determine the application class being used.

The examples in this section assume the application class is '`Emacs`'.

The Emacs resources are generally set per-frame. Each Emacs frame can have its own name or the same name as another, depending on the name passed to the `make-frame` function.

You can specify resources for all frames with the syntax:

```
Emacs*parameter: value
```

or

```
Emacs*EmacsFrame.parameter:value
```

You can specify resources for a particular frame with the syntax:

```
Emacs*FRAME-NAME.parameter: value
```

29.9.1 Geometry Resources

To make the default size of all Emacs frames be 80 columns by 55 lines, do this:

```
Emacs*EmacsFrame.geometry: 80x55
```

To set the geometry of a particular frame named 'fred', do this:

```
Emacs*fred.geometry: 80x55
```

Important! Do not use the following syntax:

```
Emacs*geometry: 80x55
```

You should never use `*geometry` with any X application. It does not say "make the geometry of Emacs be 80 columns by 55 lines." It really says, "make Emacs and all subwindows thereof be 80x55 in whatever units they care to measure in." In particular, that is both telling the Emacs text pane to be 80x55 in characters, and telling the menubar pane to be 80x55 pixels, which is surely not what you want.

As a special case, this geometry specification also works (and sets the default size of all Emacs frames to 80 columns by 55 lines):

```
Emacs.geometry: 80x55
```

since that is the syntax used with most other applications (since most other applications have only one top-level window, unlike Emacs). In general, however, the top-level shell (the unmapped ApplicationShell widget named 'Emacs' that is the parent of the shell widgets that actually manage the individual frames) does not have any interesting resources on it, and you should set the resources on the frames instead.

The `-geometry` command-line argument sets only the geometry of the initial frame created by Emacs.

A more complete explanation of geometry-handling is

- The `-geometry` command-line option sets the `Emacs.geometry` resource, that is, the geometry of the ApplicationShell.
- For the first frame created, the size of the frame is taken from the ApplicationShell if it is specified, otherwise from the geometry of the frame.
- For subsequent frames, the order is reversed: First the frame, and then the ApplicationShell.
- For the first frame created, the position of the frame is taken from the ApplicationShell (`Emacs.geometry`) if it is specified, otherwise from the geometry of the frame.
- For subsequent frames, the position is taken only from the frame, and never from the ApplicationShell.

This is rather complicated, but it does seem to provide the most intuitive behavior with respect to the default sizes and positions of frames created in various ways.

29.9.2 Iconic Resources

Analogous to `-geometry`, the `-iconic` command-line option sets the iconic flag of the ApplicationShell (`Emacs.iconic`) and always applies to the first frame created regardless of its name. However, it is possible to set the iconic flag on particular frames (by name) by using the `Emacs*FRAME-NAME.iconic` resource.

29.9.3 Resource List

Emacs frames accept the following resources:

`geometry` (class `Geometry`): string
: Initial geometry for the frame. See Section 29.9.1 [Geometry Resources], page 284, for a complete discussion of how this works.

`iconic` (class `Iconic`): boolean
: Whether this frame should appear in the iconified state.

`internalBorderWidth` (class `InternalBorderWidth`): int
: How many blank pixels to leave between the text and the edge of the window.

`interline` (class `Interline`): int
: How many pixels to leave between each line (may not be implemented).

`menubar` (class `Menubar`): boolean
: Whether newly-created frames should initially have a menubar. Set to true by default.

`initiallyUnmapped` (class `InitiallyUnmapped`): boolean
: Whether XEmacs should leave the initial frame unmapped when it starts up. This is useful if you are starting XEmacs as a server (e.g. in conjunction with gnuserv or the external client widget). You can also control this with the `-unmapped` command-line option.

`barCursor` (class `BarColor`): boolean
: Whether the cursor should be displayed as a bar, or the traditional box.

`cursorColor` (class `CursorColor`): color-name
: The color of the text cursor.

`scrollBarWidth` (class `ScrollBarWidth`): integer
: How wide the vertical scrollbars should be, in pixels; 0 means no vertical scrollbars. You can also use a resource specification of the form `*scrollbar.width`, or the usual toolkit scrollbar resources: `*XmScrollBar.width` (Motif), `*XlwScrollBar.width` (Lucid), or `*Scrollbar.thickness` (Athena). We don't recommend that you use the toolkit resources, though, because they're dependent on how exactly your particular build of XEmacs was configured.

`scrollBarHeight` (class `ScrollBarHeight`): integer
: How high the horizontal scrollbars should be, in pixels; 0 means no horizontal scrollbars. You can also use a resource specification of the form `*scrollbar.height`, or the usual toolkit scrollbar resources: `*XmScrollBar.height` (Motif), `*XlwScrollBar.height` (Lucid), or `*Scrollbar.thickness` (Athena). We don't recommend that you use the toolkit resources, though, because they're dependent on how exactly your particular build of XEmacs was configured.

`scrollBarPlacement` (class `ScrollBarPlacement`): string
: Where the horizontal and vertical scrollbars should be positioned. This should be one of the four strings '`BOTTOM_LEFT`', '`BOTTOM_RIGHT`', '`TOP_LEFT`', and '`TOP_RIGHT`'. Default is '`BOTTOM_RIGHT`' for the Motif and Lucid scrollbars and '`BOTTOM_LEFT`' for the Athena scrollbars.

`topToolBarHeight` (class `TopToolBarHeight`): integer
`bottomToolBarHeight` (class `BottomToolBarHeight`): integer
`leftToolBarWidth` (class `LeftToolBarWidth`): integer
`rightToolBarWidth` (class `RightToolBarWidth`): integer
: Height and width of the four possible toolbars.

`topToolBarShadowColor` (class `TopToolBarShadowColor`): color-name
`bottomToolBarShadowColor` (class `BottomToolBarShadowColor`): color-name
: Color of the top and bottom shadows for the toolbars. NOTE: These resources do *not* have anything to do with the top and bottom toolbars (i.e. the toolbars at the top and bottom of the frame)! Rather, they affect the top and bottom shadows around the edges of all four kinds of toolbars.

`topToolBarShadowPixmap` (class `TopToolBarShadowPixmap`): pixmap-name
`bottomToolBarShadowPixmap` (class `BottomToolBarShadowPixmap`): pixmap-name
: Pixmap of the top and bottom shadows for the toolbars. If set, these resources override the corresponding color resources. NOTE: These resources do *not* have anything to do with the top and bottom toolbars (i.e. the toolbars at the top and bottom of the frame)! Rather, they affect the top and bottom shadows around the edges of all four kinds of toolbars.

`toolBarShadowThickness` (class `ToolBarShadowThickness`): integer
: Thickness of the shadows around the toolbars, in pixels.

`visualBell` (class `VisualBell`): boolean
: Whether XEmacs should flash the screen rather than making an audible beep.

`bellVolume` (class `BellVolume`): integer
: Volume of the audible beep.

`useBackingStore` (class `UseBackingStore`): boolean
: Whether XEmacs should set the backing-store attribute of the X windows it creates. This increases the memory usage of the X server but decreases the amount of X traffic necessary to update the screen, and is useful when the connection to the X server goes over a low-bandwidth line such as a modem connection.

Emacs devices accept the following resources:

`textPointer` (class `Cursor`): cursor-name
: The cursor to use when the mouse is over text. This resource is used to initialize the variable `x-pointer-shape`.

`selectionPointer` (class `Cursor`): cursor-name
: The cursor to use when the mouse is over a selectable text region (an extent with the '`highlight`' property; for example, an Info cross-reference). This resource is used to initialize the variable `x-selection-pointer-shape`.

`spacePointer` (class `Cursor`): cursor-name
: The cursor to use when the mouse is over a blank space in a buffer (that is, after the end of a line or after the end-of-file). This resource is used to initialize the variable `x-nontext-pointer-shape`.

`modeLinePointer` (class `Cursor`): cursor-name
: The cursor to use when the mouse is over a modeline. This resource is used to initialize the variable `x-mode-pointer-shape`.

`gcPointer` (class `Cursor`): cursor-name
: The cursor to display when a garbage-collection is in progress. This resource is used to initialize the variable `x-gc-pointer-shape`.

`scrollbarPointer` (class `Cursor`): cursor-name
: The cursor to use when the mouse is over the scrollbar. This resource is used to initialize the variable `x-scrollbar-pointer-shape`.

`pointerColor` (class `Foreground`): color-name
`pointerBackground` (class `Background`): color-name
: The foreground and background colors of the mouse cursor. These resources are used to initialize the variables `x-pointer-foreground-color` and `x-pointer-background-color`.

29.9.4 Face Resources

The attributes of faces are also per-frame. They can be specified as:

```
Emacs.FACE_NAME.parameter: value
```

or

```
Emacs*FRAME_NAME.FACE_NAME.parameter: value
```

Faces accept the following resources:

`attributeFont` (class `AttributeFont`): font-name
: The font of this face.

`attributeForeground` (class `AttributeForeground`): color-name
`attributeBackground` (class `AttributeBackground`): color-name
: The foreground and background colors of this face.

`attributeBackgroundPixmap` (class `AttributeBackgroundPixmap`): file-name
: The name of an XBM file (or XPM file, if your version of Emacs supports XPM), to use as a background stipple.

`attributeUnderline` (class `AttributeUnderline`): boolean
: Whether text in this face should be underlined.

All text is displayed in some face, defaulting to the face named `default`. To set the font of normal text, use `Emacs*default.attributeFont`. To set it in the frame named `fred`, use `Emacs*fred.default.attributeFont`.

These are the names of the predefined faces:

`default`
: Everything inherits from this.

`bold`
: If this is not specified in the resource database, Emacs tries to find a bold version of the font of the default face.

`italic`
: If this is not specified in the resource database, Emacs tries to find an italic version of the font of the default face.

`bold-italic`
: If this is not specified in the resource database, Emacs tries to find a bold-italic version of the font of the default face.

`modeline`
: This is the face that the modeline is displayed in. If not specified in the resource database, it is determined from the default face by reversing the foreground and background colors.

`highlight`
: This is the face that highlighted extents (for example, Info cross-references and possible completions, when the mouse passes over them) are displayed in.

`left-margin`
`right-margin`
: These are the faces that the left and right annotation margins are displayed in.

`zmacs-region`
: This is the face that mouse selections are displayed in.

`isearch`
: This is the face that the matched text being searched for is displayed in.

`info-node`
: This is the face of info menu items. If unspecified, it is copied from `bold-italic`.

`info-xref`
: This is the face of info cross-references. If unspecified, it is copied from `bold`. (Note that, when the mouse passes over a cross-reference, the cross-reference's face is determined from a combination of the `info-xref` and `highlight` faces.)

Other packages might define their own faces; to see a list of all faces, use any of the interactive face-manipulation commands such as `set-face-font` and type '`?`' when you are prompted for the name of a face.

If the `bold`, `italic`, and `bold-italic` faces are not specified in the resource database, then XEmacs attempts to derive them from the font of the default face. It can only succeed at this if you have specified the default font using the XLFD (X Logical Font Description) format, which looks like

```
*-courier-medium-r-*-*-*-120-*-*-*-*-*-*
```

If you use any of the other, less strict font name formats, some of which look like

```
lucidasanstypewriter-12
fixed
9x13
```

then XEmacs won't be able to guess the names of the bold and italic versions. All X fonts can be referred to via XLFD-style names, so you should use those forms. See the man pages for '`X(1)`', '`xlsfonts(1)`', and '`xfontsel(1)`'.

29.9.5 Widgets

There are several structural widgets between the terminal EmacsFrame widget and the top level ApplicationShell; the exact names and types of these widgets change from release to release (for example, they changed between 19.8 and 19.9, 19.9 and 19.10, and 19.10 and 19.12) and are subject to further change in the future, so you should avoid mentioning them in your resource database. The above-mentioned syntaxes should be forward- compatible. As of 19.13, the exact widget hierarchy is as follows:

```
INVOCATION-NAME            "shell"       "container"     FRAME-NAME
x-emacs-application-class  "EmacsShell"  "EmacsManager"  "EmacsFrame"
```

where INVOCATION-NAME is the terminal component of the name of the XEmacs executable (usually '`xemacs`'), and '`x-emacs-application-class`' is generally '`Emacs`'.

29.9.6 Menubar Resources

As the menubar is implemented as a widget which is not a part of XEacs proper, it does not use the fac" mechanism for specifying fonts and colors: It uses whatever resources are appropriate to the type of widget which is used to implement it.

If Emacs was compiled to use only the Motif-lookalike menu widgets, then one way to specify the font of the menubar would be

```
Emacs*menubar*font: *-courier-medium-r-*-*-*-120-*-*-*-*-*-*
```

If the Motif library is being used, then one would have to use

```
Emacs*menubar*fontList: *-courier-medium-r-*-*-*-120-*-*-*-*-*-*
```

because the Motif library uses the `fontList` resource name instead of `font`, which has subtly different semantics.

The same is true of the scrollbars: They accept whichever resources are appropriate for the toolkit in use.

30 Correcting Mistakes (Yours or Emacs's)

If you type an Emacs command you did not intend, the results are often mysterious. This chapter discusses how you can undo your mistake or recover from a mysterious situation. Emacs bugs and system crashes are also considered.

30.1 Quitting and Aborting

C-g Quit. Cancel running or partially typed command.

C-] Abort innermost recursive editing level and cancel the command which invoked it (`abort-recursive-edit`).

M-x top-level
Abort all recursive editing levels that are currently executing.

C-x u Cancel an already-executed command, usually (`undo`).

There are two ways of cancelling commands which are not finished executing: *quitting* with *C-g*, and *aborting* with *C-]* or *M-x top-level*. Quitting is cancelling a partially typed command or one which is already running. Aborting is getting out of a recursive editing level and cancelling the command that invoked the recursive edit.

Quitting with *C-g* is used for getting rid of a partially typed command or a numeric argument that you don't want. It also stops a running command in the middle in a relatively safe way, so you can use it if you accidentally start executing a command that takes a long time. In particular, it is safe to quit out of killing; either your text will *all* still be there, or it will *all* be in the kill ring (or maybe both). Quitting an incremental search does special things documented under searching; in general, it may take two successive *C-g* characters to get out of a search. *C-g* works by setting the variable `quit-flag` to `t` the instant *C-g* is typed; Emacs Lisp checks this variable frequently and quits if it is non-`nil`. *C-g* is only actually executed as a command if it is typed while Emacs is waiting for input.

If you quit twice in a row before the first *C-g* is recognized, you activate the "emergency escape" feature and return to the shell. See Section 30.2.5 [Emergency Escape], page 293.

You can use *C-]* (`abort-recursive-edit`) to get out of a recursive editing level and cancel the command which invoked it. Quitting with *C-g* does not do this, and could not do this because it is used to cancel a partially typed command *within* the recursive editing level. Both operations are useful. For example, if you are in the Emacs debugger (see Section 23.5 [Lisp Debug], page 195) and have typed *C-u 8* to enter a numeric argument, you can cancel that argument with *C-g* and remain in the debugger.

The command *M-x top-level* is equivalent to "enough" *C-]* commands to get you out of all the levels of recursive edits that you are in. *C-]* only 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 28.5 [Recursive Edit], page 253.

`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 49.

30.2 Dealing With Emacs Trouble

This section describes various conditions in which Emacs fails to work, and how to recognize them and correct them.

30.2.1 Recursive Editing Levels

Recursive editing levels are important and useful features of Emacs, but they can seem like malfunctions to the user who does not understand them.

If the mode line has square brackets '`[...]`' around the parentheses that contain the names of the major and minor modes, you have entered a recursive editing level. If you did not do this on purpose, or if you don't understand what that means, you should just get out of the recursive editing level. To do so, type `M-x top-level`. This is called getting back to top level. See Section 28.5 [Recursive Edit], page 253.

30.2.2 Garbage on the Screen

If the data on the screen looks wrong, the first thing to do is see whether the text is actually wrong. Type `C-l`, to redisplay the entire screen. If the text 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, the real problem 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 lack specific features.

30.2.3 Garbage in the Text

If `C-l` shows that the text is wrong, try undoing the changes to it using `C-x u` until it gets back to a state you consider correct. Also try `C-h l` to find out what command you typed to produce the observed results.

If a large portion of text appears to be missing at the beginning or end of the buffer, check for the word '`Narrow`' in the mode line. If it appears, the text is still present, but marked off-limits. To make it visible again, type `C-x n w`. See Section 28.3 [Narrowing], page 252.

30.2.4 Spontaneous Entry to Incremental Search

If Emacs spontaneously displays '`I-search:`' at the bottom of the screen, it means that the terminal is sending *C-s* and *C-q* according to the poorly designed xon/xoff "flow control" protocol. You should try to prevent this by putting the terminal in a mode where it will not use flow control, or by giving it enough padding that it will never send a *C-s*. If that cannot be done, you must tell Emacs to expect flow control to be used, until you can get a properly designed terminal.

Information on how to do these things can be found in the file '`INSTALL`' in the Emacs distribution.

30.2.5 Emergency Escape

Because at times there have been bugs causing Emacs to loop without checking `quit-flag`, a special feature causes Emacs to be suspended immediately if you type a second *C-g* while the flag is already set, so you can always get out of XEmacs. 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, 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, but is just being slow, you may invoke the double *C-g* feature without really meaning to. In that case, simply resume and answer *n* to both questions, and you will arrive at your former state. Presumably the quit you requested will happen soon.

The double-*C-g* feature may be turned off when Emacs is running under a window system, since the window system always enables you to kill Emacs or to create another window and run another program.

30.2.6 Help for Total Frustration

If using Emacs (or something else) becomes terribly frustrating and none of the techniques described above solve the problem, Emacs can still help you.

First, if the Emacs you are using is not responding to commands, type `C-g C-g` to get out of it and then start a new one.

Second, type `M-x doctor` ⟨RET⟩.

The doctor will make you feel better. Each time you say something to the doctor, you must end it by typing ⟨RET⟩ ⟨RET⟩. This lets the doctor know you are finished.

30.3 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.

30.3.1 When Is There a Bug

If Emacs executes an illegal instruction, or dies with an operating system error message that indicates a problem in the program (as opposed to something like "disk full"), then it is certainly a bug.

If Emacs updates the display in a way that does not correspond to what is in the buffer, then it is certainly a bug. If a command seems to do the wrong thing but the problem corrects itself if you type `C-l`, it is a case of incorrect display updating.

Taking forever to complete a command can be a bug, but you must make certain that it was really Emacs's fault. Some commands simply take a long time. Type `C-g` and then `C-h l` to see whether the input Emacs received was what you intended to type; if the input was such that you *know* it should have been processed quickly, report a bug. If you don't know whether the command should take a long time, find out by looking in the manual or by asking for assistance.

If a command you are familiar with causes an Emacs error message in a case where its usual definition ought to be reasonable, it is probably a bug.

If a command does the wrong thing, that is a bug. But be sure you know for certain what it ought to have done. If you aren't familiar with the command, or don't know for certain how the command is supposed to work, then it might actually be working right. Rather than jumping to conclusions, show the problem to someone who knows for certain.

Finally, a command's intended definition may not be best for editing with. This is a very important sort of problem, but it is also a matter of judgment. Also, it is easy to come to such a conclusion out of ignorance of some of the existing features. It is probably best not to complain about such a problem until you have checked the documentation in the usual ways, feel confident that you understand it, and know for certain that what you want is not available. If you are not sure what the command is supposed to do after a careful reading of the manual, check the index and glossary for any terms that may be unclear. If you still do not understand, this indicates a bug in the manual. The manual's job is to make everything clear. It is just as important to report documentation bugs as program bugs.

If the online documentation string of a function or variable disagrees with the manual, one of them must be wrong, so report the bug.

30.3.2 How to Report a Bug

When you decide that there is a bug, it is important to report it and to report it in a way which is useful. What is most useful is an exact description of what commands you type, starting with the shell command to run Emacs, until the problem happens. Always include the version number of Emacs that you are using; type `M-x emacs-version` to print this.

The most important principle in reporting a bug is to report *facts*, not hypotheses or categorizations. It is always easier to report the facts, but people seem to prefer to strain to posit explanations and report them instead. If the explanations are based on guesses about how Emacs is implemented, they will be useless; we will have to try to figure out what the facts must have been to lead to such speculations. Sometimes this is impossible. But in any case, it is unnecessary work for us.

For example, suppose that you type `C-x C-f /glorp/baz.ugh` ⟨RET⟩, visiting a file which (you know) happens to be rather large, and Emacs prints out '`I feel pretty today`'. The best way to report the bug is with a sentence like the preceding one, because it gives all the facts and nothing but the facts.

Do not assume that the problem is due to the size of the file and say, "When I visit a large file, Emacs prints out '`I feel pretty today`'." This is what we mean by "guessing explanations". The problem is just as likely to be due to the fact that there is a '`z`' in the file name. If this is so, then when we got your report, we would try out the problem with some "large file", probably with no '`z`' in its name, and not find anything wrong. There is no way in the world that we could guess that we should try visiting a file with a '`z`' in its name.

Alternatively, the problem might be due to the fact that the file starts with exactly 25 spaces. For this reason, you should make sure that you inform us of the exact contents of any file that is needed to reproduce the bug. What if the problem only occurs when you have typed the `C-x a l` command previously? This is why we ask you to give the exact sequence of characters you typed since starting to use Emacs.

You should not even say "visit a file" instead of `C-x C-f` unless you *know* that it makes no difference which visiting command is used. Similarly, rather than saying "if I have three characters on the line," say "after I type ⟨RET⟩ `A B C` ⟨RET⟩ `C-p`," if that is the way you entered the text.

If you are not in Fundamental mode when the problem occurs, you should say what mode you are in.

If the manifestation of the bug is an Emacs error message, it is important to report not just the text of the error message but a backtrace showing how the Lisp program in Emacs arrived at the error. To make the backtrace, you must execute the Lisp expression `(setq debug-on-error t)` before the error happens (that is to say, you must execute that expression and then make the bug happen). This causes the Lisp debugger to run (see Section 23.5 [Lisp Debug], page 195). The debugger's backtrace can be copied as text into the bug report. This use of the debugger is possible only if you know how to make the bug happen again. Do note the error message the first time the bug happens, so if you can't make it happen again, you can report at least that.

Check whether any programs you have loaded into the Lisp world, including your '.emacs' file, set any variables that may affect the functioning of Emacs. Also, see whether the problem happens in a freshly started Emacs without loading your '.emacs' file (start Emacs with the -q switch to prevent loading the init file). If the problem does *not* occur then, it is essential that we know the contents of any programs that you must load into the Lisp world in order to cause the problem to occur.

If the problem does depend on an init file or other Lisp programs that are not part of the standard Emacs system, then you should make sure it is not a bug in those programs by complaining to their maintainers first. After they verify that they are using Emacs in a way that is supposed to work, they should report the bug.

If you can tell us a way to cause the problem without visiting any files, please do so. This makes it much easier to debug. If you do need files, make sure you arrange for us to see their exact contents. For example, it can often matter whether there are spaces at the ends of lines, or a newline after the last line in the buffer (nothing ought to care whether the last line is terminated, but tell that to the bugs).

The easy way to record the input to Emacs precisely is to write a dribble file; execute the Lisp expression:

```
(open-dribble-file "~/dribble")
```

using *Meta*-ESC or from the '*scratch*' buffer just after starting Emacs. From then on, all Emacs input will be written in the specified dribble file until the Emacs process is killed.

For possible display bugs, it is important to report the terminal type (the value of environment variable TERM), the complete termcap entry for the terminal from '/etc/termcap' (since that file is not identical on all machines), and the output that Emacs actually sent to the terminal. The way to collect this output is to execute the Lisp expression:

```
(open-termscript "~/termscript")
```

using *Meta*-ESC or from the '*scratch*' buffer just after starting Emacs. From then on, all output from Emacs to the terminal will be written in the specified termscript file as well, until the Emacs process is killed. If the problem happens when Emacs starts up, put this expression into your '.emacs' file so that the termscript file will be open when Emacs displays the screen for the first time. Be warned: it is often difficult, and sometimes impossible, to fix a terminal-dependent bug without access to a terminal of the type that stimulates the bug.

The newsgroup 'comp.emacs.xemacs' may be used for bug reports, other discussions and requests for assistance.

If you don't have access to this newgroup, you can subscribe to the mailing list version: the newsgroup is bidirectionally gatewayed into the mailing list 'xemacs@xemacs.org'.

To be added or removed from this mailing list, send mail to '`xemacs-request@xemacs.org`'. Do not send requests for addition to the mailing list itself.

The mailing lists and newsgroups are archived on our anonymous FTP server, '`ftp.xemacs.org`', and at various other archive sites around the net. You should also check the '`FAQ`' in '`/pub/xemacs`' on our anonymous FTP server. It provides some introductory information and help for initial configuration problems.

XEmacs Features

This section describes the difference between Emacs Version 18 and XEmacs.

General Changes

- XEmacs has a new vi emulation mode called "viper" mode. To start viper mode in XEmacs, type the command *`M-x viper-mode`*. If you want XEmacs to automatically put you in viper-mode all the time, include this line in your '`.emacs`' file:

  ```
  (viper-mode)
  ```

- Earlier versions of Emacs only allowed keybindings to ASCII character sequences. XEmacs has greatly expanded this by allowing you to use a vector of key sequences which are in turn composed of a modifier and a keysym. See Chapter 2 [Keystrokes], page 19 for more information.
- The keymap data structure has been reimplemented to allow the use of a character set larger than ASCII. Keymaps are no longer alists and/or vectors; they are a new primary data type. Consequently, code which manipulated keymaps with list or array manipulation functions will no longer work. It must use the functions `define-key` or `map-keymap` and `set-keymap-parent` (the new keymap functions). See Section 29.4 [Key Bindings], page 269 for more information.
- Input and display of all ISO-8859-1 characters is supported.
- Multiple fonts, including variable-width fonts, and fonts of differing heights, are supported.
- There is a new '`tags`' package and a new UNIX manual browsing package. They are similar to earlier versions; for more information look at the source code.
- There is a new implementation of Dired, with many new features. The online info for Dired, *not* the Dired node of Emacs info, provides more detail.
- GNUS (a network news reader), VM (an alternative mail reader), ILISP (a package for interacting with inferior Lisp processes), ANGE-FTP (a package for making FTP-accessible files appear just like files on the local disk, even to Dired), Calendar (an Emacs-based calendar and appointment- management tool), and W3 (an interface to the World Wide Web) are a part of the XEmacs Lisp library. See the related documentation in the online info browser.
- Emacs now supports floating-point numbers.
- When you send mail, mail aliases are now expanded in the buffer. In earlier versions, they were expanded after the mail-sending command was executed.
- The initial value of `load-path` is computed when Emacs starts up, instead of being hardcoded in when Emacs is compiled. As a result, you can now move the Emacs executable and Lisp library to a different location in the file system without having to recompile.
- Any existing subdirectories of the Emacs Lisp directory are now added to the `load-path` by default.
- On some machines, you can change the audible bell using the `sound-alist` variable. See Section 29.7 [Audible Bell], page 280 for more information.
- You can use multiple X windows to display multiple Emacs frames.
- You can use the X selection mechanism to copy material from other applications and into other applications. You can also use all Emacs region commands on a region selected with the mouse. See Section 9.2 [Mouse Selection], page 66 for more information.
- By default, the variable `zmacs-regions` is set to highlight the region between point and the mark. This unifies X selection and Emacs selection behavior.

- XEmacs has a menu bar for mouse-controlled operations in addition to keystrokes. See Section 2.4 [Pull-down Menus], page 25.
- You can look in the file '/usr/local/lib/xemacs-19.11/etc/Emacs.ad' for a list of Emacs X resources. You can set these resources in your X environment to set your preferences for color, fonts, location, and the size of XEmacs frames. Refer to your X documentation for more information about resources.

New Commands and Variables

There are many new functions in XEmacs, and many existing functions whose semantics have been expanded. Most of these are only of interest to the Emacs-Lisp programmer; see the NEWS file *C-h n* for a complete list. What follows is a partial list of the new interactive commands:

- `byte-compile-and-load-file` and `byte-compile-buffer` byte-compile the contents of a file or buffer.

  The new `conx` function lets you generate random sentences for your amusement.
- `compile-defun` compiles and evaluates the current top-level form.
- `find-this-file` and `find-this-file-other-window` can be used interactively with a prefix argument to switch to the filename at point in the buffer. `find-this-file-other-window` displays the file in another window.
- Several new functions have been added that allow you to customize the color and font attributes of a region of text: `invert-face`, `make-face-bold`, `make-face-bold-italic`, `make-face-italic`, `make-face-unbold`, `make-face-unitalic`, `set-face-background`, `set-face-background-pixmap`, `set-face-font`, `set-face-foreground`, and `set-face-underline-p`.
- `load-default-sounds` and `load-sound-file` allow you to customize the audible bell sound. `load-default-sounds` loads and installs sound files. `load-sound-file` reads in audio files and adds them to the sound alist. `play-sound` plays the specified sound type.
- `locate-library` finds the file that the function `load-library` loads, and it displays the file's full pathname.
- `make-directory` creates a directory, while `remove-directory` removes a directory.
- `mark-beginning-of-buffer` and `mark-end-of-buffer` push the mark to the beginning or end of a buffer, respectively.

  Several functions have been added that allow you to perform various editing, region, and window operations using the mouse: `mouse-del-char`, `mouse-delete-window`, `mouse-keep-one-window`, `mouse-kill-line`, `mouse-line-length`, `mouse-scroll`, `mouse-select`, `mouse-select-and-split`, `mouse-set-mark`, `mouse-set-point`, `mouse-track`, `mouse-track-adjust`, `mouse-track-and-copy-to-cutbuffer`, `mouse-track-delete-and-insert`, `mouse-track-insert`, and `mouse-window-to-region`.
- `compare-windows` takes an argument *ignore-whitespace*. The argument means ignore changes in whitespace.

You can conditionalize your '.emacs' file as follows so that XEmacs commands are invoked only when you are in XEmacs:

```
(cond ((string-match "Lucid" emacs-version)
       ;;
       ;; Code for any version of Lucid Emacs or XEmacs goes here
       ;;
       ))
```

```
(cond ((and (string-match "XEmacs" emacs-version)
    (or (> emacs-major-version 19)
        (>= emacs-minor-version 12)))
       ;;
       ;; Code which requires XEmacs version 19.12 or newer goes here
       ;;
       ))

(cond ((>= emacs-major-version 19)
       ;;
       ;; Code for any vintage-19 emacs goes here
       ;;
       ))

(cond ((and (not (string-match "Lucid" emacs-version))
    (= emacs-major-version 19))
       ;;
       ;; Code specific to FSF Emacs 19 (not XEmacs) goes here
       ;;
       ))

(cond ((< emacs-major-version 19)
       ;;
       ;; Code specific to emacs 18 goes here
       ;;
       ))
```

Of particular interest for use in '.emacs' files are:

- add-menu lets you add a new menu to the menubar or a submenu to a pull-down menu. add-menu-item, disable-menu-item, delete-menu-item, enable-menu-item, and relabel-menu-item allow you to customize the XEmacs pull-down menus.
- make-frame creates a new Emacs frame (X window).

These new variables are only present in XEmacs:

- minibuffer-confirm-incomplete prompts for confirmation in contexts where completing-read allows answers that are not valid completions.
- Several variables have been added that allow you to customize the color and shape of the mouse pointer: x-pointer-background-color, x-pointer-foreground-color, x-mode-pointer-shape, x-pointer-shape, and x-nontext-pointer-shape.
- zmacs-regions determines whether LISPM-style active regions should be used.

Changes in Key Bindings

XEmacs has the following new default function keybindings:

⟨HELP⟩ Same as *C-h*.

⟨UNDO⟩ Same as *M-x undo*.

⟨CUT⟩ Same as the Cut menu item; that is, it copies the selected text to the X Clipboard selection.

⟨COPY⟩ Same as the Copy menu item.

⟨PASTE⟩ Same as the Paste menu item.

⟨PGUP⟩ Same as *M-v*.

⟨PGDN⟩ Same as *C-v*.

⟨HOME⟩ Same as *M-<*.

⟨END⟩ Same as *M->*.

⟨LEFT-ARROW⟩
Same as the function `backward-char`.

⟨RIGHT-ARROW⟩
Same as the function `forward-char`.

⟨UP-ARROW⟩
Same as the function `previous-line`.

⟨DOWN-ARROW⟩
Same as the function `next-line`.

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 203.

Aborting Aborting means getting out of a recursive edit (q.v.). You can use the commands `C-]` and `M-x top-level` for this. See Section 30.1 [Quitting], page 291.

Auto Fill mode
Auto Fill mode is a minor mode in which text you insert is automatically broken into lines of fixed width. See Section 21.6 [Filling], page 159.

Auto Saving
Auto saving means that Emacs automatically stores the contents of an Emacs buffer in a specially-named file so 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 109.

Backup File
A backup file records the contents that a file had before the current editing session. Emacs creates backup files automatically to help you track down or cancel changes you later regret. See Section 15.3.1 [Backup], page 105.

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 22.5 [Matching Parens], page 171).

Bind To bind a key is to change its binding (q.v.). See Section 29.4.2 [Rebinding], page 271.

Binding A key gets its meaning in Emacs by having a binding which is a command (q.v.), a Lisp function that is run when the key is typed. See Section 2.3 [Commands], page 24. Customization often involves rebinding a character to a different command function. The bindings of all keys are recorded in the keymaps (q.v.). See Section 29.4.1 [Keymaps], page 270.

Blank Lines
Blank lines are lines that contain only whitespace. Emacs has several commands for operating on the blank lines in a 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 buffers can be visible when you are using multiple windows. See Chapter 16 [Buffers], page 125.

Buffer Selection History
Emacs keeps a buffer selection history which records how recently each Emacs buffer was selected. Emacs uses this list when choosing a buffer to select. See Chapter 16 [Buffers], page 125.

C- '`C`' in the name of a character is an abbreviation for Control. See Chapter 2 [Keystrokes], page 19.

C-M- '`C-M-`' in the name of a character is an abbreviation for Control-Meta. See Chapter 2 [Keystrokes], page 19.

Case Conversion
Case conversion means changing text from upper case to lower case or vice versa. See Section 21.7 [Case], page 162, for the commands for case conversion.

Characters
: Characters form the contents of an Emacs buffer; also, Emacs commands are invoked by keys (q.v.), which are sequences of one or more characters. See Chapter 2 [Keystrokes], page 19.

Command
: A command is a Lisp function specially defined to be able to serve as a key binding in Emacs. When you type a key (q.v.), Emacs looks up its binding (q.v.) in the relevant keymaps (q.v.) to find the command to run. See Section 2.3 [Commands], page 24.

Command Name
: A command name is the name of a Lisp symbol which is a command (see Section 2.3 [Commands], page 24). You can invoke any command by its name using *M-x* (see Chapter 7 [M-x], page 57).

Comments
: A comment is text in a program which is intended only for the people reading the program, and is marked specially so that it will be ignored when the program is loaded or compiled. Emacs offers special commands for creating, aligning, and killing comments. See Section 22.6 [Comments], page 172.

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 23.3 [Lisp Libraries], page 191) and programs in C and other languages (see Section 23.1 [Compilation], page 189).

Complete Key
: A complete key is a character or sequence of characters which, when typed by the user, fully specifies one action to be performed by Emacs. For example, *X* and *Control-f* and *Control-x m* are keys. Keys derive their meanings from being bound (q.v.) to commands (q.v.). Thus, *X* is conventionally bound to a command to insert 'X' in the buffer; *C-x m* is conventionally bound to a command to begin composing a mail message. See Chapter 2 [Keystrokes], page 19.

Completion
: When Emacs automatically fills an abbreviation for a name into the entire name, that process is called completion. 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 you type ⟨TAB⟩, ⟨SPC⟩, or ⟨RET⟩. See Section 6.3 [Completion], page 53.

Continuation Line
: When a line of text is longer than the width of the frame, 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 41.

Control-Character
: ASCII characters with octal codes 0 through 037, and also code 0177, do not have graphic images assigned to them. These are the control characters. Any control character can be typed by holding down the ⟨CTRL⟩ key and typing some other character; some have special keys on the keyboard. ⟨RET⟩, ⟨TAB⟩, ⟨ESC⟩, ⟨LFD⟩, and ⟨DEL⟩ are all control characters. See Chapter 2 [Keystrokes], page 19.

Copyleft
: A copyleft is a notice giving the public legal permission to redistribute a program or other work of art. Copylefts are used by leftists to enrich the public just as copyrights are used by rightists to gain power over the public.

Current Buffer
: The current buffer in Emacs is the Emacs buffer on which most editing commands operate. You can select any Emacs buffer as the current one. See Chapter 16 [Buffers], page 125.

Current Line
: The line point is on (see Section 1.1 [Point], page 14).

Current Paragraph
: The paragraph that point is in. If point is between paragraphs, the current paragraph is the one that follows point. See Section 21.4 [Paragraphs], page 158.

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 22.3 [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 41.

Customization
: Customization is making minor changes in the way Emacs works. It is often done by setting variables (see Section 29.2 [Variables], page 257) or by rebinding keys (see Section 29.4.1 [Keymaps], page 270).

Default Argument
: The default for an argument is the value that is used if you do not specify one. When Emacs prompts you in the minibuffer for an argument, the default argument is used if you just type ⟨RET⟩. See Chapter 6 [Minibuffer], page 51.

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 51.

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 22.3 [Defuns], page 166.

⟨DEL⟩
: The ⟨DEL⟩ character runs the command that deletes one character of text. See Chapter 4 [Basic Editing], page 41.

Deletion
: Deleting text means erasing it without saving it. Emacs deletes text only when it is expected not to be worth saving (all whitespace, or only one character). The alternative is killing (q.v.). See Section 10.1 [Killing], page 69.

Deletion of Files
: Deleting a file means removing it from the file system. See Section 15.10 [Misc File Ops], page 123.

Deletion of Messages
: Deleting a message means flagging it to be eliminated from your mail file. Until the mail file is expunged, you can undo this by undeleting the message.

Deletion of Frames
: When working under the multi-frame X-based version of XEmacs, you can delete individual frames using the **Close** menu item from the **File** menu.

Deletion of Windows
: When you delete a subwindow of an Emacs frame, you eliminate it from the frame. Other windows expand to use up the space. The deleted window can never come back, but no actual text is lost. See Chapter 17 [Windows], page 131.

Directory
: Files in the Unix file system are grouped into file directories. See Section 15.7 [Directories], page 119.

Dired
: Dired is the Emacs facility that displays the contents of a file directory and allows you to "edit the directory", performing operations on the files in the directory. See Section 15.9 [Dired], page 120.

Disabled Command
: A disabled command is one that you may not run without special confirmation. Commands are usually disabled because they are confusing for beginning users. See Section 29.4.3 [Disabling], page 274.

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 30.3 [Bugs], page 294.

Echo Area
: The area at the bottom of the Emacs frame which is used for echoing the arguments to commands, for asking questions, and for printing brief messages (including error messages). See Section 1.2 [Echo Area], page 14.

Echoing
: Echoing refers to acknowledging the receipt of commands by displaying them (in the echo area). Emacs never echoes single-character keys; longer keys echo only if you pause while typing them.

Error
: An error occurs when an Emacs command cannot execute in the current circumstances. When an error occurs, execution of the command stops (unless the command has been programmed to do otherwise) and Emacs reports the error by printing an error message (q.v.). Type-ahead is discarded. Then Emacs is ready to read another editing command.

Error Messages
: Error messages are single lines of output printed by Emacs when the user asks for something impossible to do (such as killing text forward when point is at the end of the buffer). They appear in the echo area, accompanied by a beep.

⟨ESC⟩
: ⟨ESC⟩ is a character used 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 and released, and applies to the next character typed.

Fill Prefix
: The fill prefix is a string that Emacs enters at the beginning of each line when it performs filling. It is not regarded as part of the text to be filled. See Section 21.6 [Filling], page 159.

Filling
: Filling text means moving text from line to line so that all the lines are approximately the same length. See Section 21.6 [Filling], page 159.

Frame
: When running Emacs on a TTY terminal, "frame" means the terminal's screen. When running Emacs under X, you can have multiple frames, each corresponding to a top-level X window and each looking like the screen on a TTY. Each frame contains one or more non-overlapping Emacs windows (possibly with associated scrollbars, under X), an echo area, and (under X) possibly a menubar.

Global
: Global means 'independent of the current environment; in effect throughout Emacs'. It is the opposite of local (q.v.). 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 203.

Global Keymap
: The global keymap (q.v.) contains key bindings that are in effect unless local key bindings in a major mode's local keymap (q.v.) override them.See Section 29.4.1 [Keymaps], page 270.

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 92.

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 29.2 [Variables], page 257.

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.) character 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 41.

Grinding
: Grinding means adjusting the indentation in a program to fit the nesting structure. See Chapter 20 [Indentation], page 145.

Hardcopy
: Hardcopy means printed output. Emacs has commands for making printed listings of text in Emacs buffers. See Section 28.4 [Hardcopy], page 253.

⟨HELP⟩
: You can type ⟨HELP⟩ at any time to ask what options you have, or to ask what any command does. ⟨HELP⟩ is really `Control-h`. See Chapter 8 [Help], page 59.

Inbox
: An inbox is a file in which mail is delivered by the operating system. Some mail handlers transfers mail from inboxes to mail files (q.v.) in which the mail is then stored permanently or until explicitly deleted.

Indentation
: Indentation means blank space at the beginning of a line. Most programming languages have conventions for using indentation to illuminate the structure of the program, and Emacs has special features to help you set up the correct indentation. See Chapter 20 [Indentation], page 145.

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 21.6 [Filling], page 159.

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 29.3 [Keyboard Macros], page 267.

Key
: A key is a sequence of characters that, when input to Emacs, specify or begin to specify a single action for Emacs to perform. That is, the sequence is considered a single unit. If the key is enough to specify one action, it is a complete key (q.v.); if it is less than enough, it is a prefix key (q.v.). See Chapter 2 [Keystrokes], page 19.

Keymap
: The keymap is the data structure that records the bindings (q.v.) of keys to the commands that they run. For example, the keymap binds the character `C-n` to the command function `next-line`. See Section 29.4.1 [Keymaps], page 270.

Kill Ring
: The kill ring is the place where all text you have killed recently is saved. You can re-insert any of the killed text still in the ring; this is called yanking (q.v.). See Section 10.2 [Yanking], page 71.

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 69.

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 33.

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. Local 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 203.

Local Keymap
: A local keymap is used in a particular major mode; the key bindings (q.v.) in the current local keymap override global bindings of the same keys. See Section 29.4.1 [Keymaps], page 270.

Local Variable
: A local value of a variable (q.v.) applies to only one buffer. See Section 29.2.4 [Locals], page 264.

M-
: `M-` in the name of a character is an abbreviation for ⟨META⟩, one of the modifier keys that can accompany any character. See Chapter 2 [Keystrokes], 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 Chapter 2 [Keystrokes], page 19.

M-x
: `M-x` is the key which is used to call an Emacs command by name. You use it to call commands that are not bound to keys. See Chapter 7 [M-x], page 57.

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 211.

Major Mode
: The major modes are a mutually exclusive set of options each of which configures Emacs for editing a certain sort of text. Ideally, each programming language has its own major mode. See Chapter 19 [Major Modes], page 143.

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 the whole region, that is, all the text from point to the mark. See Chapter 9 [Mark], page 63.

Mark Ring
: The mark ring is used to hold several recent previous locations of the mark, just in case you want to move back to them. See Section 9.1.4 [Mark Ring], page 65.

Message
: See 'mail'.

Meta
: Meta is the name of a modifier bit which a command character may have. It is present in a character if the character is typed with the ⟨META⟩ key held down. Such characters are given names that start with `Meta-`. For example, `Meta-<` is typed by holding down ⟨META⟩ and at the same time typing `<` (which itself is done, on most terminals, by holding down ⟨SHIFT⟩ and typing `,`). See Chapter 2 [Keystrokes], page 19.

Meta Character
: A Meta character is one whose character code includes the Meta bit.

Minibuffer
: The minibuffer is the window that Emacs displays inside the echo area (q.v.) when it prompts you for arguments to commands. See Chapter 6 [Minibuffer], page 51.

Minor Mode
: A minor mode is an optional feature of Emacs which can be switched on or off independent of the major mode. Each minor mode has a command to turn it on or off. See Section 29.1 [Minor Modes], page 257.

Mode Line
: The mode line is the line at the bottom of each text window (q.v.), which gives status information on the buffer displayed in that window. See Section 1.3 [Mode Line], page 15.

Modified Buffer
: A buffer (q.v.) is modified if its text has been changed since the last time the buffer was saved (or since it was created, if it has never been saved). See Section 15.3 [Saving], page 104.

Moving Text
: Moving text means erasing it from one place and inserting it in another. This is done by killing (q.v.) and then yanking (q.v.). See Section 10.1 [Killing], page 69.

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 79.

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 the invisible text. See Section 28.3 [Narrowing], page 252.

Newline
: ⟨LFD⟩ characters in the buffer terminate lines of text and are called newlines. See Chapter 2 [Keystrokes], page 19.

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 46.

Option
: An option is a variable (q.v.) that allows you to customize Emacs by giving it a new value. See Section 29.2 [Variables], page 257.

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 29.1 [Minor Modes], page 257.

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 21.5 [Pages], page 158.

Paragraphs
: Paragraphs are the medium-size unit of English text. There are special Emacs commands for moving over and operating on paragraphs. See Section 21.4 [Paragraphs], page 158.

Parsing
: We say that Emacs parses words or expressions in the text being edited. Really, all it knows how to do is find the other end of a word or expression. See Section 29.5 [Syntax], page 274.

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 41.

Prefix Key
: A prefix key is a key (q.v.) whose sole function is to introduce a set of multi-character keys. *`Control-x`* is an example of a prefix key; any two-character sequence starting with *`C-x`* is also a legitimate key. See Chapter 2 [Keystrokes], page 19.

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 51); the echoing which happens when you pause in the middle of typing a multi-character key is also a kind of prompting (see Section 1.2 [Echo Area], page 14).

Quitting
: Quitting means cancelling a partially typed command or a running command, using *`C-g`*. See Section 30.1 [Quitting], page 291.

Quoting
: Quoting means depriving a character of its usual special significance. In Emacs this is usually done with *`Control-q`*. What constitutes special significance depends on the context and on convention. For example, an "ordinary" character as an Emacs command inserts itself; so in this context, a special character is any character that does not normally insert itself (such as ⟨DEL⟩, for example), and quoting it makes it insert itself as if it were not special. Not all contexts allow quoting. See Chapter 4 [Basic Editing], page 41.

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, such asDired buffers. Visiting a file that is write-protected also makes a read-only buffer. See Chapter 16 [Buffers], page 125.

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 28.5 [Recursive Edit], page 253.

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 [Frame], page 13.

Regexp
: See 'regular expression'.

Region
: The region is the text between point (q.v.) and the mark (q.v.). Many commands operate on the text of the region. See Chapter 9 [Mark], page 63.

Registers
: Registers are named slots in which text or buffer positions or rectangles can be saved for later use. See Chapter 11 [Registers], page 79.

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 89.

Replacement
: See 'global substitution'.

Restriction
: A buffer's restriction is the amount of text, at the beginning or the end of the buffer, that is temporarily invisible and inaccessible. Giving a buffer a nonzero amount of restriction is called narrowing (q.v.). See Section 28.3 [Narrowing], page 252.

⟨RET⟩
: ⟨RET⟩ is the character than 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 Chapter 2 [Keystrokes], page 19.

Saving
: Saving a buffer means copying its text into the file that was visited (q.v.) in that buffer. To actually change a file you have edited in Emacs, you have to save it. See Section 15.3 [Saving], page 104.

Scrolling
: Scrolling means shifting the text in the Emacs window to make a different part ot the buffer visible. See Chapter 12 [Display], page 81.

Searching
: Searching means moving point to the next occurrence of a specified string. See Chapter 13 [Search], page 85.

Selecting
: Selecting a buffer means making it the current (q.v.) buffer. See Chapter 16 [Buffers], page 125.

Self-documentation
: Self-documentation is the feature of Emacs which can tell you what any command does, or can 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 59.

Sentences
: Emacs has commands for moving by or killing by sentences. See Section 21.3 [Sentences], page 157.

Sexp
: An sexp (short for 's-expression,' itself short for 'symbolic 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. If simultaneous editing is not detected, you may lose your work. Emacs detects all cases of simultaneous editing and warns the user to investigate them. See Section 15.3.2 [Simultaneous Editing], page 107.

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. Write a '`"`' that is part of the string as '`\"`' and a '`\`' that is part of the string as '`\\`'. You can include all other characters, including newline, just by writing them inside the string. You can also include escape sequences as in C, such as '`\n`' for newline or '`\241`' using an octal character code.

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 29.5 [Syntax], page 274.

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 22.11 [Tags], page 176.

Termscript File
: A termscript file contains a record of all characters Emacs sent to the terminal. It is used for tracking down bugs in Emacs redisplay. Emacs does not make a termscript file unless explicitly instructed to do so. See Section 30.3 [Bugs], page 294.

Text
: Text has two meanings (see Chapter 21 [Text], page 149):

- 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 something that follows 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 30.1 [Quitting], page 291.

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 97).

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 41.

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 49.

Variable
: A variable is Lisp object that can store an arbitrary value. Emacs uses some variables for internal purposes, and has others (known as 'options' (q.v.)) you can set 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 of this manual. See Section 29.2 [Variables], page 257, for information on variables.

Visiting
: Visiting a file means loading its contents into a buffer (q.v.) where they can be edited. See Section 15.2 [Visiting], page 102.

Whitespace
: Whitespace is any run of consecutive formatting characters (spaces, tabs, newlines, and backspaces).

Widening
: Widening is removing any restriction (q.v.) on the current buffer; it is the opposite of narrowing (q.v.). See Section 28.3 [Narrowing], page 252.

Window
: Emacs divides the frame into one or more windows, each of which can display the contents of one buffer (q.v.) at any time. See Chapter 1 [Frame], page 13, for basic information on how Emacs uses the frame. See Chapter 17 [Windows], page 131, for commands to control the use of windows. Note that if you are running Emacs under X, terminology can be confusing: Each Emacs frame occupies a separate X window and can, in turn, be divided into different subwindows.

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 88.

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 71.

The GNU Manifesto

What's GNU? GNU's Not Unix!

GNU, which stands for GNU's Not Unix, is the name for the complete Unix-compatible software system which I am writing so that I can give it away free to everyone who can use it. Several other volunteers are helping me. Contributions of time, money, programs, and equipment are greatly needed.

So far we have an Emacs text editor with Lisp for writing editor commands, a source level debugger, a yacc-compatible parser generator, a linker, and around 35 utilities. A shell (command interpreter) is nearly completed. A new portable optimizing C compiler has compiled itself and may be released this year. An initial kernel exists, but many more features are needed to emulate Unix. When the kernel and compiler are finished, it will be possible to distribute a GNU system suitable for program development. We will use TeX as our text formatter, but an nroff is being worked on. We will use the free, portable X window system as well. After this we will add a portable Common Lisp, an Empire game, a spreadsheet, and hundreds of other things, plus online documentation. We hope to supply, eventually, everything useful that normally comes with a Unix system, and more.

GNU will be able to run Unix programs, but will not be identical to Unix. We will make all improvements that are convenient, based on our experience with other operating systems. In particular, we plan to have longer filenames, file version numbers, a crashproof file system, filename completion perhaps, terminal-independent display support, and perhaps eventually a Lisp-based window system through which several Lisp programs and ordinary Unix programs can share a screen. Both C and Lisp will be available as system programming languages. We will try to support UUCP, MIT Chaosnet, and Internet protocols for communication.

GNU is aimed initially at machines in the 68000/16000 class with virtual memory, because they are the easiest machines to make it run on. The extra effort to make it run on smaller machines will be left to someone who wants to use it on them.

To avoid horrible confusion, please pronounce the 'G' in the word 'GNU' when it is the name of this project.

Why I Must Write GNU

I consider that the golden rule requires that if I like a program I must share it with other people who like it. Software sellers want to divide the users and conquer them, making each user agree not to share with others. I refuse to break solidarity with other users in this way. I cannot in good conscience sign a nondisclosure agreement or a software license agreement. For years I worked within the Artificial Intelligence Lab to resist such tendencies and other inhospitalities, but eventually they had gone too far: I could not remain in an institution where such things are done for me against my will.

So that I can continue to use computers without dishonor, I have decided to put together a sufficient body of free software so that I will be able to get along without any software that is not free. I have resigned from the AI lab to deny MIT any legal excuse to prevent me from giving GNU away.

Why GNU Will Be Compatible With Unix

Unix is not my ideal system, but it is not too bad. The essential features of Unix seem to be good ones, and I think I can fill in what Unix lacks without spoiling them. And a system compatible with Unix would be convenient for many other people to adopt.

How GNU Will Be Available

GNU is not in the public domain. Everyone will be permitted to modify and redistribute GNU, but no distributor will be allowed to restrict its further redistribution. That is to say, proprietary modifications will not be allowed. I want to make sure that all versions of GNU remain free.

Why Many Other Programmers Want to Help

I have found many other programmers who are excited about GNU and want to help.

Many programmers are unhappy about the commercialization of system software. It may enable them to make more money, but it requires them to feel in conflict with other programmers in general rather than feel as comrades. The fundamental act of friendship among programmers is the sharing of programs; marketing arrangements now typically used essentially forbid programmers to treat others as friends. The purchaser of software must choose between friendship and obeying the law. Naturally, many decide that friendship is more important. But those who believe in law often do not feel at ease with either choice. They become cynical and think that programming is just a way of making money.

By working on and using GNU rather than proprietary programs, we can be hospitable to everyone and obey the law. In addition, GNU serves as an example to inspire and a banner to rally others to join us in sharing. This can give us a feeling of harmony which is impossible if we use software that is not free. For about half the programmers I talk to, this is an important happiness that money cannot replace.

How You Can Contribute

I am asking computer manufacturers for donations of machines and money. I'm asking individuals for donations of programs and work.

One consequence you can expect if you donate machines is that GNU will run on them at an early date. The machines should be complete, ready-to-use systems, approved for use in a residential area, and not in need of sophisticated cooling or power.

I have found very many programmers eager to contribute part-time work for GNU. For most projects, such part-time distributed work would be very hard to coordinate; the independently-written parts would not work together. But for the particular task of replacing Unix, this problem is absent. A complete Unix system contains hundreds of utility programs, each of which is documented separately. Most interface specifications are fixed by Unix compatibility. If each contributor can write a compatible replacement for a single Unix utility, and make it work properly in place of the original on a Unix system, then these utilities will work right when put together. Even allowing for Murphy to create a few unexpected problems, assembling these components will be a feasible task. (The kernel will require closer communication and will be worked on by a small, tight group.)

If I get donations of money, I may be able to hire a few people full or part time. The salary won't be high by programmers' standards, but I'm looking for people for whom building community spirit is as important as making money. I view this as a way of enabling dedicated people to devote their full energies to working on GNU by sparing them the need to make a living in another way.

Why All Computer Users Will Benefit

Once GNU is written, everyone will be able to obtain good system software free, just like air.

This means much more than just saving everyone the price of a Unix license. It means that much wasteful duplication of system programming effort will be avoided. This effort can go instead into advancing the state of the art.

Complete system sources will be available to everyone. As a result, a user who needs changes in the system will always be free to make them himself, or hire any available programmer or company to make them for him. Users will no longer be at the mercy of one programmer or company which owns the sources and is in sole position to make changes.

Schools will be able to provide a much more educational environment by encouraging all students to study and improve the system code. Harvard's computer lab used to have the policy that no program could be installed on the system if its sources were not on public display, and upheld it by actually refusing to install certain programs. I was very much inspired by this.

Finally, the overhead of considering who owns the system software and what one is or is not entitled to do with it will be lifted.

Arrangements to make people pay for using a program, including licensing of copies, always incur a tremendous cost to society through the cumbersome mechanisms necessary to figure out how much (that is, which programs) a person must pay for. And only a police state can force everyone to obey them. Consider a space station where air must be manufactured at great cost: charging each breather per liter of air may be fair, but wearing the metered gas mask all day and all night is intolerable even if everyone can afford to pay the air bill. And the TV cameras everywhere to see if you ever take the mask off are outrageous. It's better to support the air plant with a head tax and chuck the masks.

Copying all or parts of a program is as natural to a programmer as breathing, and as productive. It ought to be as free.

Some Easily Rebutted Objections to GNU's Goals

> "Nobody will use it if it is free, because that means they can't rely on any support."

> "You have to charge for the program to pay for providing the support."

If people would rather pay for GNU plus service than get GNU free without service, a company to provide just service to people who have obtained GNU free ought to be profitable.

We must distinguish between support in the form of real programming work and mere hand-holding. The former is something one cannot rely on from a software vendor. If your problem is not shared by enough people, the vendor will tell you to get lost.

If your business needs to be able to rely on support, the only way is to have all the necessary sources and tools. Then you can hire any available person to fix your problem; you are not at the mercy of any individual. With Unix, the price of sources puts this out of consideration for most businesses. With GNU this will be easy. It is still possible for there to be no available competent person, but this problem cannot be blamed on distibution arrangements. GNU does not eliminate all the world's problems, only some of them.

Meanwhile, the users who know nothing about computers need handholding: doing things for them which they could easily do themselves but don't know how.

Such services could be provided by companies that sell just hand-holding and repair service. If it is true that users would rather spend money and get a product with service, they will also be willing to buy the service having got the product free. The service companies will compete in quality and price; users will not be tied to any particular one. Meanwhile, those of us who don't need the service should be able to use the program without paying for the service.

> "You cannot reach many people without advertising, and you must charge for the program to support that."

> "It's no use advertising a program people can get free."

There are various forms of free or very cheap publicity that can be used to inform numbers of computer users about something like GNU. But it may be true that one can reach more microcomputer users with advertising. If this is really so, a business which advertises the service of copying and mailing GNU for a fee ought to be successful enough to pay for its advertising and more. This way, only the users who benefit from the advertising pay for it.

On the other hand, if many people get GNU from their friends, and such companies don't succeed, this will show that advertising was not really necessary to spread GNU. Why is it that free market advocates don't want to let the free market decide this?

"My company needs a proprietary operating system to get a competitive edge."

GNU will remove operating system software from the realm of competition. You will not be able to get an edge in this area, but neither will your competitors be able to get an edge over you. You and they will compete in other areas, while benefitting mutually in this one. If your business is selling an operating system, you will not like GNU, but that's tough on you. If your business is something else, GNU can save you from being pushed into the expensive business of selling operating systems.

I would like to see GNU development supported by gifts from many manufacturers and users, reducing the cost to each.

"Don't programmers deserve a reward for their creativity?"

If anything deserves a reward, it is social contribution. Creativity can be a social contribution, but only in so far as society is free to use the results. If programmers deserve to be rewarded for creating innovative programs, by the same token they deserve to be punished if they restrict the use of these programs.

"Shouldn't a programmer be able to ask for a reward for his creativity?"

There is nothing wrong with wanting pay for work, or seeking to maximize one's income, as long as one does not use means that are destructive. But the means customary in the field of software today are based on destruction.

Extracting money from users of a program by restricting their use of it is destructive because the restrictions reduce the amount and the ways that the program can be used. This reduces the amount of wealth that humanity derives from the program. When there is a deliberate choice to restrict, the harmful consequences are deliberate destruction.

The reason a good citizen does not use such destructive means to become wealthier is that, if everyone did so, we would all become poorer from the mutual destructiveness. This is Kantian ethics; or, the Golden Rule. Since I do not like the consequences that result if everyone hoards information, I am required to consider it wrong for one to do so. Specifically, the desire to be rewarded for one's creativity does not justify depriving the world in general of all or part of that creativity.

"Won't programmers starve?"

I could answer that nobody is forced to be a programmer. Most of us cannot manage to get any money for standing on the street and making faces. But we are not, as a result, condemned to spend our lives standing on the street making faces, and starving. We do something else.

But that is the wrong answer because it accepts the questioner's implicit assumption: that without ownership of software, programmers cannot possibly be paid a cent. Supposedly it is all or nothing.

The real reason programmers will not starve is that it will still be possible for them to get paid for programming; just not paid as much as now.

Restricting copying is not the only basis for business in software. It is the most common basis because it brings in the most money. If it were prohibited, or rejected by the customer, software business would move to other bases of organization which are now used less often. There are always numerous ways to organize any kind of business.

Probably programming will not be as lucrative on the new basis as it is now. But that is not an argument against the change. It is not considered an injustice that sales clerks make the salaries that they now do. If programmers made the same, that would not be an injustice either. (In practice they would still make considerably more than that.)

"Don't people have a right to control how their creativity is used?"

"Control over the use of one's ideas" really constitutes control over other people's lives; and it is usually used to make their lives more difficult.

People who have studied the issue of intellectual property rights carefully (such as lawyers) say that there is no intrinsic right to intellectual property. The kinds of supposed intellectual property rights that the government recognizes were created by specific acts of legislation for specific purposes.

For example, the patent system was established to encourage inventors to disclose the details of their inventions. Its purpose was to help society rather than to help inventors. At the time, the life span of 17 years for a patent was short compared with the rate of advance of the state of the art. Since patents are an issue only among manufacturers, for whom the cost and effort of a license agreement are small compared with setting up production, the patents often do not do much harm. They do not obstruct most individuals who use patented products.

The idea of copyright did not exist in ancient times, when authors frequently copied other authors at length in works of non-fiction. This practice was useful, and is the only way many authors' works have survived even in part. The copyright system was created expressly for the purpose of encouraging authorship. In the domain for which it was invented—books, which could be copied economically only on a printing press—it did little harm, and did not obstruct most of the individuals who read the books.

All intellectual property rights are just licenses granted by society because it was thought, rightly or wrongly, that society as a whole would benefit by granting them. But in any particular situation, we have to ask: are we really better off granting such license? What kind of act are we licensing a person to do?

The case of programs today is very different from that of books a hundred years ago. The fact that the easiest way to copy a program is from one neighbor to another, the fact that a program has both source code and object code which are distinct, and the fact that a program is used rather than read and enjoyed, combine to create a situation in which a person who enforces a copyright is harming society as a whole both materially and spiritually; in which a person should not do so regardless of whether the law enables him to.

"Competition makes things get done better."

The paradigm of competition is a race: by rewarding the winner, we encourage everyone to run faster. When capitalism really works this way, it does a good job; but its defenders are wrong in assuming it always works this way. If the runners forget why the reward is offered and become intent on winning, no matter how, they may find other strategies—such as, attacking other runners. If the runners get into a fist fight, they will all finish late.

Proprietary and secret software is the moral equivalent of runners in a fist fight. Sad to say, the only referee we've got does not seem to object to fights; he just regulates them ("For every ten yards you run, you can fire one shot"). He really ought to break them up, and penalize runners for even trying to fight.

"Won't everyone stop programming without a monetary incentive?"

Actually, many people will program with absolutely no monetary incentive. Programming has an irresistible fascination for some people, usually the people who are best at it. There is no shortage of professional musicians who keep at it even though they have no hope of making a living that way.

But really this question, though commonly asked, is not appropriate to the situation. Pay for programmers will not disappear, only become less. So the right question is, will anyone program with a reduced monetary incentive? My experience shows that they will.

For more than ten years, many of the world's best programmers worked at the Artificial Intelligence Lab for far less money than they could have had anywhere else. They got many kinds of non-monetary rewards: fame and appreciation, for example. And creativity is also fun, a reward in itself.

Then most of them left when offered a chance to do the same interesting work for a lot of money.

What the facts show is that people will program for reasons other than riches; but if given a chance to make a lot of money as well, they will come to expect and demand it. Low-paying organizations do poorly in competition with high-paying ones, but they do not have to do badly if the high-paying ones are banned.

> "We need the programmers desperately. If they demand that we stop helping our neighbors, we have to obey."

You're never so desperate that you have to obey this sort of demand. Remember: millions for defense, but not a cent for tribute!

> "Programmers need to make a living somehow."

In the short run, this is true. However, there are plenty of ways that programmers could make a living without selling the right to use a program. This way is customary now because it brings programmers and businessmen the most money, not because it is the only way to make a living. It is easy to find other ways if you want to find them. Here are a number of examples.

A manufacturer introducing a new computer will pay for the porting of operating systems onto the new hardware.

The sale of teaching, hand-holding, and maintenance services could also employ programmers.

People with new ideas could distribute programs as freeware and ask for donations from satisfied users or sell 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 a certain 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.

Key (Character) Index

Command and Function Index

A

B

C

D

J

K

L

M

U

V

W

X

Y

Z

Variable Index

Z

Concept Index

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

Short Contents

Table of Contents