Go to the previous, next section.

Searching and Replacement

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

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

Incremental Search

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

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

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

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

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

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

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

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

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

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

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

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

To search for a control character such as `C-s` or `DEL` or `ESC`, you must quote it by typing `C-q` first. This function of `C-q` is analogous to its meaning as an Emacs command: it causes the following character to be treated the way a graphic character would normally be treated in the same context. You can also specify a quoted character in octal while searching, just as you can for insertion. See section Basic Editing Commands.

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

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

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

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

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

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

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

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

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

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

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

Slow Terminal Incremental Search

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

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

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

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

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

Nonincremental Search

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

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

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

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

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

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

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

Word Search

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

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

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

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

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

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

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

Regular Expression Search

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

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

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

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

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

Syntax of Regular Expressions

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

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

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

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

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

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

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

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

`+`

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

`?`

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

`[ ... ]`

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

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

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

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

`[^ ... ]`

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

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

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

`^`

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

`$`

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

`\`

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

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

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

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

`\|`

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

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

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

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

`\( ... \)`

is a grouping construct that serves three purposes:

To enclose a set of `` `\|' `` alternatives for other operations. Thus, `` `\(foo\|bar\)x' `` matches either `` `foox' `` or `` `barx' ``.

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.

To mark a matched substring for future reference.

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

`\`*`digit`*

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

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

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

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

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

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

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

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

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

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

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

`\s`*code*
: matches any character whose syntax is *code*. *code* is a character which represents a syntax code: thus, `` `w' `` for word constituent, `` `-' `` for whitespace, `` `(' `` for open-parenthesis, etc. See section The Syntax Table.

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

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

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

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

Note that the above example shows how to write this regexp when entering it as part of an Emacs Lisp program. To enter the same regexp in an interactive command such as `re-search-forward` you must spell it differently:

```
[.?!][]"')]*\($\|^Q^I\|  \)[ ^Q^I^Q^J]*
```

Searching and Case

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

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

Replacement Commands

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

The replace commands all replace one string (or regexp) with one replacement string. It is possible to perform several replacements in parallel using the command `expand-region-abbrevs`. See section Controlling Abbrev Expansion.

Unconditional Replacement

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

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

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

Narrowing).

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.

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 `` `\\' ``.

Replace Commands and Case

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

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

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

Query Replace

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

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

If you want to change only some of the occurrences of `` `foo' `` to `` `bar' ``, not all of them, then you cannot use an ordinary `replace-string`. Instead, use `M-%` (`query-replace`). This command finds occurrences of

`` `foo' `` one by one, displays each occurrence and asks you whether to replace it. A numeric argument to `query-replace` tells it to consider only occurrences that are bounded by word-delimiter characters.

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

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

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

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

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

`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 Recursive Editing Levels.

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

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

`C-h`

to display a message summarizing these options, then give another answer.

If you type any other character, the `query-replace` is exited, and the character executed as a command. To restart the `query-replace`, use `C-x ESC`, which repeats the `query-replace` because it used the minibuffer to read its arguments. See section Repeating Minibuffer Commands.

To replace every occurrence, you can start `query-replace` at the beginning of the buffer and type `!`, or you can use the `replace-string` command at the beginning of the buffer. To replace every occurrence in a part of the buffer, narrow to that part and then run `replace-string` or `query-replace` at the beginning of it. See section Narrowing.

Other Search-and-Loop Commands

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

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

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

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

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

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

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

Go to the previous, next section.