A Collection of Tutorials on Emacs

A Companion to the FAQ and the Emacs Manual

Kai Grossjohann `Kai.Grossjohann@CS.Uni-Dortmund.DE`

Table of Contents

A Collection of Tutorials on Emacs

This collection of tutorials is not complete. I use narrow paragraphs prefixed with “CCC” to point out open issues. Don’t hesitate to contact me if you have an idea about one of them.

CCC TODO

- Skeletons

1 Indentation

1.1 Basic Concepts

There are quite a few concepts in Emacs which relate to white space at the beginning of a line. Enough, in fact, to be potentially quite confusing. I'll try to clear up the confusion in this little document. The following concepts come into play:

1.1.1 The ⟨TAB⟩ key.

Contrary to popular belief, the ⟨TAB⟩ key does not insert a tab character in most modes. Instead, it does syntax-driven indentation in most programming language modes. In fundamental mode, it advances to the next tab stop. In text mode, it does relative indentation (relative to the previous line, that is). You can type `C-q` ⟨TAB⟩ to insert a tab character.

1.1.2 The tab character.

It is possible to change the way a tab character is displayed. '`vi`' users do this. For example, they can set the width of a tab character to 4, and this enables them to hit ⟨TAB⟩ at the beginning of a line to indent the body of a loop, say, by one indentation step (of 4 characters). Normally, the width of tab is eight characters, so a file created by a '`vi`' user with a tab width of 4 will look different when viewed in a program which uses the default tab width setting.

In order to correctly view files with non-standard tab width settings, it is possible to set the variable `tab-width`, like this:

```
(setq-default tab-width 4)
```

If you only want this in a particular mode, add `(setq tab-width 4)` to that mode's hook. (See below for an explanation of mode hooks.)

1.1.3 Whether or not tab characters are used for indentation.

Whether by syntax-driven indentation, or relative indentation (text mode), or advancing to the next tab stop (fundamental mode), indentation needs to insert white space. The variable `indent-tabs-mode` controls whether or not tab characters are used for inserting the white space. If this variable is `t`, tab characters are used to make the file as short as possible. For example, if `tab-width` is equal to 4 and white space of 10 characters needs to be inserted at the beginning of the line, Emacs will insert 2 tab characters (2 times 4 equals 8 spaces), plus two space characters. With `tab-width` equal to the default value of 8, Emacs would insert 1 tab plus 2 spaces.

Use the following line to tell Emacs to never use tab characters for indentation:

```
(setq-default indent-tabs-mode nil)
```

Use `t` rather than `nil` to tell Emacs to use tab characters where appropriate.

If you only want this in a particular mode, add `(setq indent-tabs-mode nil)` to that mode's hook. (See below for an explanation of mode hooks.)

By the way, actual tab characters are really needed in Makefiles; thus, makefile mode overrides this setting.

1.1.4 The location of the tab stops.

The variable `tab-stop-list` contains a list of tab stop positions, and *M-i* moves to the next tab stop. (In fundamental mode, ⟨TAB⟩ also does this.) You can define your own list with a statement like the following:

```
(setq tab-stop-list '(4 8 12 16))
```

Whether or not actual tab characters are inserted is controlled by the variable `indent-tabs-mode`, and the number of tab characters inserted is controlled by `tab-width` (among others).

Obviously, if you set `tab-stop-list` to a list containing multiples of `tab-width`, then one *M-i* will insert one tab character at all times (if `indent-tabs-mode` isn't `nil`).

The variable `tab-stop-list` controls which spots *M-i* moves to, and `indent-tabs-mode` controls whether tab characters (plus maybe a few space characters) are used to move to that spot, and `tab-width` controls how many tab characters are needed.

1.1.5 Syntax driven indentation.

This means that the current line is indented in a way that is useful for the syntax of the programming language used. For example, in C-like languages, the lines between an opening and a closing brace are usually indented two to four spaces with respect to the braces themselves. In Pascal, the lines between `begin` and `end` would be indented two to four spaces.

Since the syntax driven indentation depends a lot on the syntax of the programming language at hand, the major mode provides the indentation. And customizing each major mode works a little differently, because the programming languages are different.

For C-like languages, there is '`CC mode`'. It provides major modes for C, C++, Objective C, IDL, and Java. The indentation engine is quite flexible, and the CC mode info file has the whole story on customizing indentation for these major modes. I'll just mention a few things.

CC mode provides for different coding styles. Type *M-x c-set-style* ⟨RET⟩, then use tab completion to have a look at them, and try them out. If you find one you like, you can add something like `(c-set-style "gnu")` to your mode hook. These coding styles often refer to a basic indentation offset, determined by the variable `c-basic-offset`. Add stuff like `(setq c-basic-offset 4)` to the mode hook in question.

If this isn't sufficient, you can go to the line where you don't like the indentation, and type *C-c C-o*. Follow the prompts. Hit ⟨TAB⟩ to re-indent the current line. Repeat until you like the result. You can then type *C-x* ⟨ESC⟩ ⟨ESC⟩ and use *M-n* and *M-p* to locate the right '`c-set-offset`' statement to add to your hook. Please note that you can enter numbers for the indentation level, but you can also use `+` and `-` to stand for one basic offset to the right or left, respectively, or `++` and `--` for twice the basic offset.

I use '`CPerl`' mode for editing Perl code. It also provides a style mechanism similar to the one provided by CC mode, but it doesn't provide an interactive command which allows one to customize indentation, like *C-c C-o* in CC mode. Type *C-h f cperl-mode* ⟨RET⟩ for a list of variables affecting indentation. Choose an indentation style which is closest to what you want, then change the remaining variables from your mode hook. You can choose an indentation style by adding something like `(cperl-set-style "gnu")` to your mode hook.

CCC Can people provide info for other modes?

1.1.6 Relative indentation.

This means indent to a spot indicated by the previous non-blank line. Below, the spots relative indentation moves to are indicated with a caret.

```
This is a line to show relative indentation.
^    ^  ^ ^    ^  ^    ^        ^
```

I find this useful for text where it can be used to format tables or to indent itemized lists. When in text mode, the ⟨TAB⟩ key performs relative indentation by default. The command name is `indent-relative`.

1.2 Usage scenarios.

In the above, I explained the concepts behind the things Emacs is doing, but I suspect you'd like some advice on when to frob which options.

1.2.1 I want to turn off syntax driven indentation!

Since '`vi`' doesn't have syntax driven indentation, its users usually set the tab width to some value they like, and whenever they want to indent a line more than the previous one, they hit ⟨TAB⟩. Of course, a file which was edited in this way will look strange when loaded into Emacs. The variable `tab-width` was made for this, just set it to the tab width your '`vi`' loving colleagues use, and you can continue to use syntax driven indentation and your cow-orkers will wonder why you can edit code so fast. . . .

Perhaps you also want to set `indent-tabs-mode` to `t` (which is the default) such that Emacs actually uses tabs with the new width. But then, maybe you want to set it to `nil` such that Emacs uses only spaces and the looks of your code won't depend on the tab width setting somebody uses.

And you might need to customize your indentation to be similar to the style used by the others, lest you need to do a lot of manual reformatting. A consistent coding style is a good thing.

1.3 Background information.

This section provides background information referred to in the above.

1.3.1 Mode hooks.

A central concept in Emacs is the *major mode*. For each kind of text, a major mode provides functions useful in that mode. Most (if not all) major modes provide a so-called *hook* which allows you to specify Lisp code to be run when that major mode is entered. Most of the time, this feature is used just for setting a few variables.

A *mode hook* is a variable which contains a list of functions to be executed. Thus, if you want something to happen, you have to write a function first. Writing functions is rather simple, finding a name for the function might be the most difficult part, in fact!

So, let's say you wish to set the variable `c-basic-offset` to 3 when in C++ mode. Additionally, you want to use no tabs for indentation. Here's a function which does this:

```
(defun my-c++-indent-setup ()
  (setq c-basic-offset 3)
  (setq indent-tabs-mode nil))
```

Pay careful attention to the nesting of the parentheses. The empty pair of parentheses is always needed, and the parenthesis opened before the word `defun` is closed at the very end of the function.

So, all that's left is to add this function to the mode hook for C++ mode. Easy:

```
(add-hook 'c++-mode-hook 'my-c++-indent-setup)
```

You may wish to replace the '`my`' prefix with something else such that you can be sure not to reuse a function name already defined elsewhere. I use the prefix '`kai`', for example. This means I've got to be careful when snarfing code from Kai-Uwe Rommel :-)

1.4 TODO

- file local variables for tab-width (Per Abrahamsen)

2 Key bindings

Binding functions to keys is something most new users of Emacs probably do rather soon. This subject has also been discussed to quite some extent in the FAQ, so be sure to look there as well as the chapter on customizing key bindings in the Emacs manual.

2.1 Online help for key bindings

A key binding is an association between a key (sequence) and a Lisp command. You can type `C-h c` *x* to find out which command is invoked by the key (sequence) *x* in the current major mode. You can get more help about *x* with `C-h k` *x*. You can type `C-h w` *f* ⟨RET⟩ to find out which key bindings exist for the given function *f*. Another very useful keybinding is `C-h b` which gives you a list of all key bindings currently in effect.

So, maybe `C-h w` tells you that the function you want already has a key binding, so you don't need your own?

2.2 A warning about own key bindings

Of course, many people prefer their own key bindings rather than the standard ones. Since no two people work alike, it is a Good Thing that Emacs allows you to choose your own bindings, of course. However, a few rules should be obeyed for choosing your own key bindings:

- All key bindings of the form `C-c` *x* where *x* is any letter are reserved for the user. This means that you can count on all well-behaved packages not to clobber these bindings. Please note that bindings for non-letters might be used by major modes. (AUC-TeX uses `C-c ;` for `comment-region`, for instance.) The bindings `C-c C-`*x* where *x* is a letter are reserved for the major modes.
- Most control key combinations (such as `C-f`) and most meta key combinations (such as `M-f`) are already used by Emacs. A few modes change the meanings of a few standard key bindings, so it is advisable to stay away from the standard key bindings in the beginning, until one gets a feel for the kinds of standard key bindings which are most useful.

  As an example, `M-a` normally moves backward a sentence, but in C mode it moves backward a statement.

  It might be best if you get into the habit of doing `C-h b` in the major modes you use.

2.3 Simple key bindings

Even though the function in question might already have a key binding, you may prefer your own, because it is more easily reachable on your keyboard, say. As an example, the following line makes it so that `C-c f` invokes `forward-paragraph` in all modes:

```
(global-set-key (kbd "C-c f") 'forward-paragraph)
```

It is also rather simple to establish key bindings for particular modes, for instance:

```
(define-key c-mode-map (kbd "C-c <f1>") 'mark-whole-buffer)
```

Some modes aren't loaded when Emacs starts. This means that Emacs might not know about the variable `c-mode-map` when it starts. A simple solution exists: just load the package before using the variable. C mode is defined in the '`CC mode`' package, so the following line helps:

```
(require 'cc-mode)
```

The normal rule is that *foo-mode* is defined in the package *foo-mode* or possibly in *foo*. The C-like modes are an exception, they are defined in the package `cc-mode`. You can type `C-h f` *foo-mode* `RET` which will tell you which package the function comes from.

2.4 The syntax for '`kbd`'

What kind of string should you use between the double quotes after '`kbd`'? Well, the syntax is not hard to figure out; explaining it is a bit more difficult. Well, anyway, here goes:

Basically, the syntax used by '`kbd`' is the same as the syntax used by Emacs to describe key bindings (in `C-h c` and `C-h b`, among other places), with one exception. The exception concerns function key names, which should be enclosed in angle brackets. Compare `(kbd "C-c <f1>")` used above: F1 is a function key and thus its name is enclosed in angle brackets. By the way, you write `C-<f1>` if you want the ⟨Ctrl⟩ key to be pressed while hitting F1.

Why aren't ⟨RET⟩ and ⟨SPC⟩ enclosed in angle brackets? Well, even though it might look as if they are function keys, we're really talking about ASCII character, here. A few ASCII characters aren't printable and thus there are three-letter names for them: ⟨RET⟩ (ASCII 13), ⟨DEL⟩ (ASCII 127), ⟨LFD⟩ (ASCII 10), ⟨SPC⟩ (ASCII 32), ⟨ESC⟩ (ASCII 27) and ⟨TAB⟩ (ASCII 9).

In general, you can tell the symbolic name for an ASCII characters from function key names by the case: function key names are lowercase, ASCII characters have uppercase names.

Which leads us to the distinction between the ASCII key ⟨RET⟩ and the function key ⟨return⟩: On a serial terminal or using a terminal emulation like '`xterm`', Emacs cannot tell the difference between the user hitting ⟨return⟩ and the user hitting `C-m`, so there is only ⟨RET⟩. Changing the key binding for ⟨RET⟩ also changes it for `C-m` and vice versa. But if you are using a windowing system (such as X11), Emacs does know the difference, and makes it available to you. One way to use it is to make `C-m` behave different from ⟨return⟩. Another area where the difference becomes important is with additional modifiers: `(kbd "C-<return>")` makes perfect sense, but `(kbd "C-RET")` is not meaningful since this would be the same as `(kbd "C-C-m")` which in turn is clearly bogus—you cannot "double-press" the control key, so `C-RET` is double-plus ungood.

Whee. More complex than you thought, right? On the other hand, most of the time you don't need to know all this—just look at the output from `C-h c` and remember to enclose function key names in angle brackets, and Bob will be your uncle. But there is one area where the previous paragraph becomes important, because these keys are so often used:

2.5 The delete and backspace keys

Let's talk about windowing systems, first. By default the ⟨delete⟩ and ⟨backspace⟩ keys do the same thing: they generate the ASCII character ⟨DEL⟩ which in turn is bound to `backward-delete-char-untabify` in many modes. But in quite a few modes, ⟨DEL⟩ has a different binding which is normally useful. For example, in view mode, ⟨DEL⟩ is page-up, which is similar to the Unix programs '`more`' and '`less`', more or less.

Most people want the ⟨backspace⟩ key to delete backwards whereas the ⟨delete⟩ key should delete forward. But how to do that without losing the useful ⟨DEL⟩ bindings? Well, there are three potential key bindings (⟨DEL⟩, ⟨backspace⟩, ⟨delete⟩) but only two keys, so what you should do is to choose the right two key bindings, and ⟨DEL⟩ should be among them. The standard semantic of ⟨DEL⟩ is vaguely "backward", so the natural thing to do is to let ⟨backspace⟩ be ⟨DEL⟩ as is the default and to change the binding of ⟨delete⟩: (global-set-key (kbd "<delete>") 'delete-char)

If you don't like what the ⟨backspace⟩ key does, don't change its binding directly. Instead, change the binding for ⟨DEL⟩: `(global-set-key (kbd "DEL") 'backward-delete-char)`

2.6 Binding *C-u something* to a different key

C-u invokes `universal-argument`, which changes the behavior of quite a few standard functions. For example, in C mode the key combination *C-c C-c* comments out the region. For *uncommenting* the region, you have to use *C-u C-c C-c*.

Thus, one question which comes up a few times is how to make the F9 key do what *C-u C-c C-c* does.

Sadly, it is not as simple as one might think. You have to use some Lisp for this. First, let's have a look what *C-c C-c* does in C mode, so we type *C-h k C-c C-c*, which produces the following help:

```
C-c C-c runs the command comment-region
   which is an interactive compiled Lisp function in `simple'.
(comment-region BEG END &optional ARG)

Comment or uncomment each line in the region.
With just C-u prefix arg, uncomment each line in region.
Numeric prefix arg ARG means use ARG comment characters.
If ARG is negative, delete that many comment characters instead.
Comments are terminated on each line, even for syntax in which
newline does not end the comment.  Blank lines do not get comments.
```

Obviously, uncommenting is the same as deleting comment characters, so the line about ARG being negative is the crucial line, here. What you want to do is to write a function which is the same as `comment-region`, except that it passes a negative ARG:

```
(defun uncomment-region (beg end)
  "Like `comment-region' invoked with a C-u prefix arg."
  (interactive "r")
  (comment-region beg end -1))
```

The part about invoking `comment-region` with `-1` as argument should be clear, but the `interactive` line looks quite obfuscated. Well, you have to use `interactive` such that a function can be invoked via *M-x* or a key binding. And the string `"r"` means that the two arguments of this function are the start and the end of the region, respectively. See the documentation of the `interactive` function for more alternatives.

You can now bind `uncomment-region` to a key, just like any other command.

2.7 Complex key bindings

There are two ways to bind complex actions to a single key. One of them is easy to use but is not easily maintainable, and the other requires a bit of programming but is more flexible. The easy way to do it is to use keyboard macros, the flexible way is to write Lisp functions.

2.7.1 Complex key bindings with kbd macros

Type *C-x (* to start recording a kbd macro, use *C-x)* to stop recording. Use *M-x name-last-kbd-macro RET* to give a name to the kbd macro just recorded. Then, use *M-x insert-kbd-macro RET* to insert the named kbd macro into your '`.emacs`' file. The name of the kbd macro can then be used just like any other command name in a normal '`define-key`' or '`global-set-key`' statement.

2.7.2 Complex key bindings with Lisp functions

Teaching Lisp is out of the scope of this tutorial, but you may wish to have a look at the Emacs Lisp Introduction, available from `ftp.gnu.org` and its mirrors, as well as the section in the Emacs manual on the '`.emacs`' file.

2.8 TODO

- Add xref to (emacs)Init File.

3 Mail with Gnus

Quite a few people wish to read mail with Gnus but stumble across a few issues which make this a bit difficult. This comes from the fact that Gnus is really a newsreader, and thus it treats mail in a newsreaderly fashion. It is not desirable to change this, because it is a wonderful thing and this is what distinguishes Gnus from other mail readers. In this little tutorial, I'll try to explain what "newsreaderly fashion" means and how exactly Gnus treats mail.

Specific pieces of behavior can always be changed, but if you desire to make Gnus behave like a conventional mail reader, think again. It will be an uphill battle. Maybe a different mail reader is for you? But also, read on. Maybe you'll find the right behavior in the description below.

3.1 Gnus Terminology

First, let's talk about a few terms which I'm going to use which might be unfamiliar to you.

Posting, Article, Message, Mail
: These are all related terms. A news message might also be called a posting or an article, whereas a mail message is known as a mail. Since Gnus treats both news and mail, the distinction isn't as clear. In the following, I'll use the term "message" for the generic case and "news message" and "mail message" for the specific cases. But sometimes, I might use "posting" or "article", too, both synonymous with "message".

Backend
: Gnus can read messages from various sources. On the one hand, there is news, and on the other hand, there is mail. News can be read from a news server (an NNTP server), or from a so-called spool directory. Mail can be read in various formats.

  Generally speaking, a backend describes the way Gnus accesses a set of messages. For a news server, this is the Network News Transfer Protocol, NNTP, and thus there is a backend "nntp". For mail stored in the same format that the MH message handling system used, there is the backend "nnmh". And so on. See below for a list of backends.

Server
: Whereas a backend describes the mechanism used by Gnus to access the messages, a server is a specific instance of that mechanism. You might create a specific server for accessing the host '`news.frob.org`' using NNTP, say. Or you might create a server for accessing the MH-style directory '`~/OldMail`'.

  If you are a programmer, think of a backend as the class and of a server as an object (instance) of that class. If you are a cook, think of a backend as an apple pie recipe (say), and think of a server as an actual apple pie. (Yummy!) If you live in a huge city, think of a backend as a bus (or tram, or underground) line (the Circle Line comes to mind), and think of a server as a specific bus (or tram train, or underground train). The one at 8:15 last Monday morning, for example. If you drive a car, think of a backend as the model and make and of a server as a specific car.

Obviously, there can be two servers using the same backend. (Two instances of the same class, two apple pies baked according to the same recipe, two busses going the same route, two cars of the same model.)

(Select) method
Just another term for server.

Native server
This is the primary server, so to speak. Most people let their news server be the native server, hence:

```
(setq gnus-select-method '(nntp "news.frob.org"))
```

Groups from the native server are also known as native groups.

Secondary select methods
This is a list of other servers which one also wishes to use. Many people are only going to have two servers, one native (for news) and one secondary (for mail). Thus:

```
(setq gnus-secondary-select-methods '((nnml "")))
```

Note that there is one more pair of parentheses in order to be able to mention more than one seconary select method.

Groups from a secondary server are also known as secondary groups.

In order to be able to distinguish native groups from secondary groups, each server is identified with a (unique) name and that name is used as a prefix for the secondary groups. Thus, you might have a group '`gnu.emacs.help`' (which is native) and another group '`nnml:mail.misc`' (which is secondary). A plus character is used if the name of a server is not the empty string. For example, given the following in your '`.gnus`' file

```
(setq gnus-secondary-select-methods
      '((nnml "work" (nnml-directory "~/Mail.work/"))
        (nnml "play" (nnml-directory "~/Mail.play/"))))
```

you might have the groups '`nnml+work:boss`' and '`nnml+play:so`'[1].

Group Well, if you've read news before, you know about different news groups. One of my favorites is '`gnu.emacs.gnus`', and I wish I would read '`alt.fan.pratchett`'. Since Gnus treats mail in a newsreaderly fashion, it is obvious that it uses groups rather than "folders" like other mail readers do. So with Gnus there are news groups and mail groups, where mail groups are known as mail folders to other programs.

Each group belongs to a certain server, and each server uses a certain backend.

Expiry News servers offer news groups which contain news postings. New news postings come in, so the news postings accumulate, and pretty soon the new hard disk is full. This is not good news at all.

Thus, a news server does what is known as *expiry*: it deletes old messages. Of course, on a news server, expiry happens with no regard of whether people have already seen the message in question; instead, the news server admin chooses

[1] "SO" standing for "significant other", i.e. girlfriend or boyfriend

expiry times based on available disk space and maybe on the normal amount of traffic in a news group.

But mail messages should be under the users' control, so there better be no server which deletes messages regardless of users having seen them! Instead, Gnus adopts a scheme where users say which messages may be deleted, and Gnus takes care of deleting them after a while. (They are not deleted immediately in case you made a mistake, or in case you wish to refer back to an old article.)

Article marks

Gnus distinguishes between a number of article marks, which indicate whether they have been looked at, or are considered important, or the like. Marks are represented by a character.

If that character is a space, it looks as if the message isn't marked at all. These messages are called *unmarked*, the mark character used is a space, and marking a message with space is considered to be the same as removing all marks—after all, such messages are unmarked. You can type `M-u` to remove all marks and make an article unmarked.

Articles that are considered important or where you wish to indicate that you have to deal with them later can be *ticked*. The mark character used for ticked messages is the exclamation mark, and you can use `u` or `!` to tick messages. Ticked messages are always shown when you enter a group.

There is the *dormant* mark which is similar to the ticked mark but does not imply importance or urgency; thus, dormant messages aren't shown by default. The mark character used is the question mark, and you can mark messages as dormant using the `?` key.

So far, each kind of mark was associated with one character (as was the absence of any mark). But articles which you have read are a bit different, since lots of different characters are used here. The important thing to realize is that these messages are treated in the same way by Gnus; the different characters are only used as an information for the user.

Articles which are marked as read because you have actually read them (the normal case, one would think) are marked with the 'R' character. (Type ⟨SPC⟩ or `g` to read a message, or click on it using the middle mouse button, ⟨mouse-2⟩.) You can also mark a message as read without actually reading it, this is indicated with the 'r' character and can be achieved with `d` or `M r`.

After exiting a group and then entering it again (some time later), the messages that had been marked as read before appear with the 'O' character.

To reiterate: the difference between 'r', 'R' and 'O' is only an information for the user.

3.2 Choosing a mail backend

The Gnus manual lists quite a few backends. Of these, the news backends pose no problem: you use the `nntp` backend if you access a news server and the `nnspool` backend if you have a local news spool directory. (Leafnode users should use `nntp` so that the leafnode program can see what you are doing and choose the groups to download accordingly.) But the mail backends are more difficult. There are many of them, and it is not obvious which is the best choice. In quite a few cases, this is because there is no single best choice; or maybe what's the best choice depends on the group or changes over time.

Below, I'll give a list of mail backends. While I say something about how messages are stored, I'll try to emphasize what that means for you as a user.

Let's try to structure the discussion a bit. We have servers, which contain groups, which in turn contain messages. How could we store this on disk? After some thought, you'll quickly come up with the following alternatives: You could store all messages from a server in one file. The second alternative is to store all messages from one group in one file, different groups are stored in different files. A third alternative is to store each message in one file; in this case, one could use a directory per group. A very interesting fourth alternative is not to store the messages at all but instead to use the Oracle of Delphi (say) to predict what the messages will be; this saves a lot of disk space. I won't talk about the fourth alternative in the following.

3.2.1 Backends with one file per server

Many people use just two servers, the native server for news and a secondary server for mail. Thus, this alternative would mean that you store all your mail in one file. Since Emacs has no fancy mechanisms to access parts of files, this means that Gnus loads that file into main memory at startup, and all your mails are kept in main memory all the time. (Of course, copies are written to disk every now and then, for safekeeping!)

I think you can pretty much figure out the consequences on your own, now:

- Handling large amounts of mail will be a problem. (Emacs has a maximum file size of 128 MB.)
- Some operations on mails will be fast, since they are in-memory operations. (Saving everything to disk will be slow, though.)
- Some operations on mails will be slow, since they have to search through the whole file.
- It is convenient to have all mail stored in one file: you can easily transfer it to another computer using FTP, say, or store it on a floppy or Zip disk or a tape.

Conclusion: If you don't have a lot of mail to deal with and like the convenience of storing it all in one file, one of these backends might be for you. However, since Gnus really shines when dealing with lots of mails, most Gnus users can be expected to deal with quite a large volume of mail. Thus, I don't think many Gnus users choose one of these backends.

`nnmbox`
: This backend uses the well-known "mbox" format for storing mails. In this format, a message begins with the five characters `From_` (the last character is a space) at the beginning of a line, and ends with an empty line.

`nnbabyl`
: This backend uses the lesser known "babyl" format for storing mails. This uses delimiters for the beginning and the end of a message which are less likely to occur in a message.

    CCC Are they guaranteed to never occur?

One advantage of a babyl file over an mbox file is that it is possible to insert information about a message in the babyl file, without having to change the message itself. In an mbox file, the only place to put such information is the message header, which is part of the message. Thus, adding information about a message to an mbox file means that one has to change the message.

I think Gnus doesn't make use of this advantage, though. Gnus stores information about messages in an extra file, '`~/.newsrc.eld`'.

CCC Can somebody provide me with some more arguments in favor of one of the formats?

CCC Is it possible to just use an existing babyl file for Gnus, by creating a new nnmbox server and pointing it at the file? What about mbox?

3.2.2 Backends with one file per group

Storing all messages in a group in one file provides a nice middle ground between the one file per server type of backend and the one file per message type of backend. Using lots of little files wastes disk space; since this approach uses a moderate number of files, less disk space is wasted.

CCC Which operations are fast using this kind of backend? Which are slow?

`nnfolder`
: This backend uses the same file format as `nnmbox`, but uses the one file per group approach.

There is no "nnbabylfolder" backend which uses babyl format.

3.2.3 Backends with one file per message

If the number of messages grows so large that even the size of a single group exceeds the limit which can be handled by the file-per-group backends, you need to think about using one of the backends mentioned here.

This category also includes `nnml`, the backend which is fastest if you have lots of messages.

`nnmh`
: This backend uses the same file format (and directory structure) as MH, i.e. a group is a directory, and each message is stored in a file, and the file names are numbers.

  Since `nnml` is so similar to `nnmh` but a lot faster, only unusual situations could warrant using this backend. You may want to use `nnmh` if you wish to use Gnus in parallel to your old MH based reader.

  Normally, you should not let two programs write the same Gnus directory (not even two instances of Gnus!), but if you really must, you may wish to use `nnmh`, since there the probability of things breaking is smaller than with the other backends.

`nnml`
: This backend is like `nnmh` but also includes an extra file '`.overview`' in each directory (group) which contains some headers from each message. Thus, where `nnmh` needs to open every file in a group to examine its headers, `nnml` (normally) needs to only read the '`.overview`' file, which is a lot faster.

3.2.4 Other mail backends

There is one other mail backend, for keeping messages on an IMAP server.

`nnimap` This backend transforms Gnus into an IMAP client. The general idea of IMAP is to store and manipulate the mails on a server (similar to NNTP for news).

`nnimap` comes with Gnus 5.8, it does not work with earlier Gnus versions. But there is also a Web site for `nnimap`: `http://www.extundo.com/nnimap/`.

3.2.5 Summary

If you must talk to an IMAP server, the choice is clear. But if you keep mails on your local disk, the choice isn't as clear-cut. I think that `nnml` is generally the best choice unless you have real great disk space trouble. Then, you should be thinking about `nnfolder`.

I'm not sure if there is a situation where `nnmbox` or `nnbabyl` is desirable.

CCC Tell me about it if you know more!

3.3 Auto-expire versus total-expire

Every now and then, people ask about auto-expire and total-expire. Since both of these features are means to the same end, and since they are similar and dissimilar at the same time, great confusion can result in the unsuspecting new Gnus user. I'll try to explain how each works and which one to use. However, be prepared that there will be no clear recommendation: both work well, so for many situations both are appropriate. So it is more a matter of taste which one to choose. And I can't help you with that!

3.3.1 What is expiry?

Gnus treats mail in a newsreaderly fashion, so it is useful to examine the situation for news. Your news server periodically contacts other news servers and exchanges messages with the other server. The two news servers exchange lists of messages, and messages present in one server but not in the other are sent to the other server. This works in both directions. Many connections between news servers exist, and this is the way how postings travel from you into the world: when you post a message, your news server stores it and offers it to the other servers in the message list exchanging phase. Since the other servers aren't going to have the posting you just wrote, it gets transferred and finally can be seen all over the world.

You can quickly see that new messages will be arriving at your news server, which stores them on disk. So something has got to happen else the disk will fill up real fast. That "something" is expiry: the oldest messages are deleted to make room for the new ones. Normally, a news server can be configured on a per-group basis which messages should be deleted. In some groups, messages might be deleted which are older than a day, in other groups, messages might be kept for a month.

This means if you go on vacation then come back later to read news, you are likely to miss some postings if the expiration time for the groups you read is shorter than the time you were on vacation.

How does that apply to mail?

Well, mail should stay more under the control of the user than news is. When you come back from a vacation, you expect to see all messages arrived during that time, not only the recent ones!

Because of this, Gnus offers you the `E` key. This marks a message as expirable. No mail is ever deleted from disk, unless it is expirable. Every once in a while (by default, whenever you quit a group by hitting `q` in the Summary buffer), the expiry process is run, which goes through all expirable messages (of a group) and expires it if old enough. By default, messages older than seven days are "old enough". Seven days, that is, since it was marked as expirable.

CCC Last sentence correct?

"But when I read a message, exit the group, then enter it again, the message is gone!"

Right. By default, Gnus hides messages which have already been read. If you are the keyboard type, you can hit `C-u` ⟨RET⟩ or `C-u` ⟨SPC⟩ to enter the group or `C-u M-g` when in the group to look at the read messages. If you are the mousey type, you may wish to use the "See old articles" entry in the "Group" menu.

CCC How does one code menu entries in TeXinfo?

3.3.2 Why auto-expire and total-expire?

When reading mail, by default Gnus marks as read each message that you read. If you want to mark it as expirable, you have to hit `E`. Many people are subscribed to mailing lists and they configure Gnus to put mails from a mailing list into their own group. Most messages in such groups should be expirable, once read. But hitting `E` all the gets old real quick. Hence, auto-expire and total-expire were invented.

3.3.3 Auto-expire vs. total-expire

Auto-expire and total-expire both aim at the same goal: articles which are read should normally be expired, only if one does something special should these articles be saved on disk. But what happens when a message is being read by you, the user? Well, the message is marked as read (with the 'R' mark). So, what can be done to make these messages expire? Well, two approaches come to mind: one idea is to change the mark that is attached to messages that you read, and the other idea is to make the 'R' articles expirable. These are *exactly* the things that are done in Gnus: auto-expire means to change the mark that is attached to a message that is being read, and total-expire means to change the meaning of the 'R' mark to mean expirable.

A more precise description of auto-expire might be the following: If an article is unmarked and then selected for reading,[2] it is marked with 'E', as if the user had hit `E`.

It is important to realize that auto-expire has *no other* consequences. Selecting a message for reading which wasn't unmarked doesn't do anything special, and hitting `d` on a message also doesn't do anything special. (It therefore marks the message as read, not as expirable!)

Now, forget about auto-expire, empty your mind and prepare to learn about total-expire. Like I said, total-expire changes what it means for an article to be marked as read.

[2] Using `g`, or ⟨mouse-2⟩, or by moving to it with `n` or `p`, or by one of the many other methods provided by Gnus.

A more precise description of total-expire might be the following: When the expire process is run (for example, when you leave a group with `q`), all messages marked as read are considered to be expirable, as if they had been marked with 'E'. Recall that there are several ways to mark a message as read: by reading it, by hitting `d` on it, and in a few other ways which I haven't mentioned so far. Recall that, in addition to the messages marked with 'R', also those marked with 'r' or 'O' are considered to be marked as read.

Can auto-expire and total-expire be used together? Well, in principle they can, but that doesn't make sense. Just using total-expire alone achieves the same effect.

So, how do auto-expire and total-expire compare? Well, for once thing, hitting `d` on a message means it is expirable if total-expire is on (since it is marked as read and all messages marked as read are considered expirable when total-expire is on), but it is not expirable if auto-expire is on (since it is marked as read and only articles marked expirable ('E') are considered to be expirable). If you want to mark a message as expirable when total-expire is off, use `E`.

One way of comparing auto-expire and total-expire is to compare the message marks that are available, and what they mean. Since auto-expire does not change the meaning of marks, its behavior is the same as in the default case. It's only important whether total-expire is on or off. Thus, there are only two cases: the default case and the total-expire case.

3.3.3.1 Article marks with and without total-expire

The following are the default article marks and behavior:

unmarked All new messages are unmarked. This means you haven't seen them. They are always shown and won't be deleted.

read Messages marked as read are not shown by default but kept on disk till hell freezes over. You can show them with `C-u M-g` from the summary buffer, or with `C-u` ⟨SPC⟩ or with the 'Group' menu item 'See old articles' from the group buffer.

Depending on the setting of *gnus-fetch-old-headers*, a message marked as read might be shown if there is a followup or reply to it.

dormant Dormant messages aren't shown by default but kept on disk till hell freezes over. You can show them with `/ D` from the summary buffer. If there is a reply or followup to a dormant message, the dormant message is also shown.

ticked Ticked messages are always shown and kept on disk till hell freezes over.

expirable Expirable messages will be deleted in due time. They are not shown by default, but you can make them appear with `C-u M-g` and so on, similar to the read ones.

Please note that the behavior for ticked messages is similar to the unread ones, and that the behavior of dormant messages is similar to the read ones. Especially the second fact will become important when we look at

The behavior of the article marks with total-expire:

unmarked Same as in the default case.

expirable Same as in the default case.

read Same as expirable.

dormant Same as in the default case.

ticked Same as in the default case.

As you can see, only the behavior of the read messages is different, and you can use the dormant mark if you want to achieve behavior similar to the behavior of read messages in the default case.

3.3.3.2 Speed issues

Total-expire may be slow when expiry happens. Why is that? Well, Gnus keeps an explicit list of all expirable messages (the ones marked 'E' without taking total-expire into account), as well as a list of dormant messages, and a list of ticked messages. Normally, when expiration time comes, Gnus goes through all articles in the expire list and looks if they are old enough to be expired.

However, for read messages the situation is different. Here, Gnus just keeps a list of ranges of article numbers to be able to distinguish read messages from unmarked ones. The assumption is that a message is to be considered marked as read if it falls in one of the ranges and isn't mentioned in any of the expirable, dormant or ticked lists.

When total-expire is turned on, Gnus needs to go through all messages in the read range in order to look if it's in one of the lists. If the message is not in the ticked or dormant list, it is expirable and thus Gnus looks to see if it is old enough.

Obviously, going through all the articles in the read ranges takes more time than going through just the list of expirable articles.

Something can be done about the speed issue, though. Normally, the expiry process is started whenever you leave a group. I suggest that you disable this and instead run expiry just once per day, for example while you are going for lunch. This means that expiry still takes a long time, but you don't see it and thus it doesn't bother you.

Here's how to do that: You disable the expire-on-group-exit thing with the following line in your '`~/.gnus`' file:

```
(remove-hook 'gnus-summary-prepare-exit-hook
             'gnus-summary-expire-articles)
```

And before you leave for lunch, you hit `C-c C-M-x`, or `M-x gnus-group-expire-all-groups` ⟨RET⟩.

3.3.3.3 Functionality

Adaptive scoring doesn't work with auto-expire. (But normal scoring still works fine.) Adaptive scoring works fine with total-expire.

3.3.3.4 A summary

Well, it is difficult to sum up the whole discussion. I used to use total-expire but have switched to auto-expire a long time ago. I liked the fact that I could use one more kind of article mark. I also liked the fact that marking a message as read works the same in auto-expirable groups and in normal mail groups: you hit *E*. (With total-expire, you normally hit *d* but must remember to use *E* for those groups where total-expire is off.) And I liked that auto-expire is faster.

On the other hand, adaptive scoring is surely a very useful feature (I'm just beginning to use it), so many people might prefer total-expire.

And on a third hand, maybe the key binding issue isn't so important after all. You see, in mail groups the *d* key means 'keep this message for archival purposes', whereas in many other modes (dired, CCC others?) it stands for 'delete'. I propose to make it mean delete in mail groups, too, with the following line in '`~/.gnus`':[3]

```
(define-key gnus-summary-mode-map "d" 'gnus-summary-mark-as-expirable)
```

Marking messages as expirable (rather than read) in news groups does no harm, nor does it harm to do so in total-expirable mail groups. The old 'keep this message' semantics can still be had by marking a message as dormant or by using *M r* (in non-total-expirable groups only).

3.4 Migrating old mail

Probably, you've been reading mail in pre-Gnus times, right? And surely you wish to carry this over to Gnus. Well, I haven't found a real good way to do it, but I can offer a few suggestions for doing it at least semi-automatically.

One way of getting at your old mail is to type *G f*, and to then enter the name of your old mail file. This provides read-only access to your mails. For some people, this might be sufficient. (With *G f*, you have created an `nndoc` group.)

Some people might want to have their mails available in their normal mail groups hierarchy. That's simple, just create an `nndoc` group for your mail file, then mark all messages in it with *M P* `a`, then copy all of them over to a normal mail group, with *B c*.

This is good for people who wish to keep their old arrangement of folders, and who have a one-to-one correspondence between old mail files and new Gnus groups. But some people might wish to split up their mails differently. For them, it might be useful to set up *nnmail-split-methods* correctly and to use *B r* instead of *B c*. This goes through all process-marked messages and subjects them to the same splitting process that newly arriving messages go through. (Whee! What a run-on sentence!)

3.5 TODO

- Say something about the cache. Though this belongs in the news reading tips, right? Hm.

[3] See the chapter on key bindings; maybe you need a 'require' statement.