

Emacs For Writers

A guide for the curious and obsessive (Version 2)

Chris Maiorana

Table of Contents

1	Introduction	1
1.1	Why Plain Text?	1
2	What is Emacs?	2
3	The Frame	3
3.1	The menu bar	3
3.2	The toolbar	3
3.3	The window	3
3.4	Mode line	4
3.5	Echo area (mini buffer)	4
4	Emacs keys and keybindings	5
4.1	Control, meta, and more	5
4.2	Keybindings and how they work	5
5	Navigation (moving around in Emacs)	7
5.1	Opening files in direcd, the directory editor	7
5.2	Navigating the buffer	7
5.3	A special note on keybindings	8
6	Introducing Org Mode	9
6.1	The outliner	9
6.2	Org Mode syntax	9
6.3	The Org Mode exporter	10
6.4	The tag system	10
7	More syntactic goodness	11
7.1	Bold, italics, preformatted text	11
7.2	Subs and supers	11
7.3	Structured blocks (quotes and verses and code)	11
8	Your first "config"	13
8.1	Emacs Lisp is your customization language	13
8.2	How to install your Emacs configuration	13
8.3	Common errors you might see on startup	14
8.3.1	No such file or directory	14
8.3.2	Void function: "Symbol's function definition is void"	14
8.3.3	End of file during parsing	14
8.3.4	Invalid read syntax	14

8.4	Troubleshooting your config file (if you happen to get errors) ...	14
8.5	Don't overdo it with your config	15
8.6	Going deep with Emacs Lisp	15
9	Installing packages	16
9.1	Setting up the package archives	16
9.2	The magic of "use-package"	16
9.3	Package refresh	17
9.4	Install packages	17
10	The magic of abbrev mode	18
10.1	Create your own abbrev definitions	18
10.1.1	Setting abbrevs with key commands	18
10.1.2	With Emacs Lisp statements	19
10.2	Saving your abbrevs into memory	19
10.3	Unexpand an abbrev	19
11	Counting words and word counters	20
11.1	Tracking lines and line numbers	20
11.2	Relative line numbers	20
12	Export	21
12.1	Some theory - filename separation	21
12.2	Using the export dispatcher	21
12.3	Using my custom function	22
12.4	Exporting your documents like a pro	22
13	Research and note-taking	23
13.1	Value of notes	23
13.2	Fishing for ideas	23
13.3	Make everything a note	23
13.3.1	A flat hierarchy	24
13.3.2	Simplicity	24
13.3.3	Necessary tools	24
13.4	"Binding" notes	25
13.4.1	Use cases	25
13.4.2	How to use links	25
13.4.3	Hot tip: saving link patterns	25
13.5	My personal note system	26
13.5.1	Unique files	26
13.5.2	Creating linked files from text	26
13.5.3	Turn snapshot into a note	27
13.5.4	Rewriting another file	27
13.5.5	Downloading and using NB	27

14	File organization theory	28
14.1	File temperature	28
14.2	File attachments	28
14.2.1	What's happening on the backend	29
14.2.2	Create a new attached file	29
14.2.3	Open the attachment directory in Dired.....	29
14.2.4	Use cases.....	29
14.3	Heading management.....	29
14.3.1	"Noexport" a whole heading.....	30
14.3.2	Ignore a heading.....	30
15	Tracking goals and other metrics	31
15.1	Tracking deadlines with the agenda.....	31
15.1.1	What is the agenda?.....	31
15.1.2	Create a deadline.....	31
15.1.3	Adding your file to the agenda.....	31
15.1.4	Warning days	32
15.2	Logging notes on headings.....	32

1 Introduction

Where does it start? Always with insatiable curiosity. I have been writing all my life. I started with a pencil, then moved up to Microsoft Word, Apple's Pages, and then I found text editors. And everything about the **way** I wrote started to change. Curiosity leads the way.

I presume you found this page because you're on a mission. You want **freedom** from expensive software licenses, bloated interfaces, and you want to write at the speed of your thoughts. You've come to the right place at just the right time.

Keep reading.

1.1 Why Plain Text?

Unlike Word Processors, which store your text in binary files, a text editor stores your words in vanilla plain text; thus, you can free yourself from being locked up in one proprietary format. Not only that, plain text is fast, easy to read, and cross-platform. A plain text document written thirty years ago is just as crisp, relevant, and open today as it was at its inception.

Text editors are most commonly associated with computer programming. The raw text is "compiled" into machine readable computer instructions.

But the freedom and portability of text editors like Emacs makes them a great alternative to conventional writing software.

If your words matter to you, plain text is a great way to store and retrieve them.

2 What is Emacs?

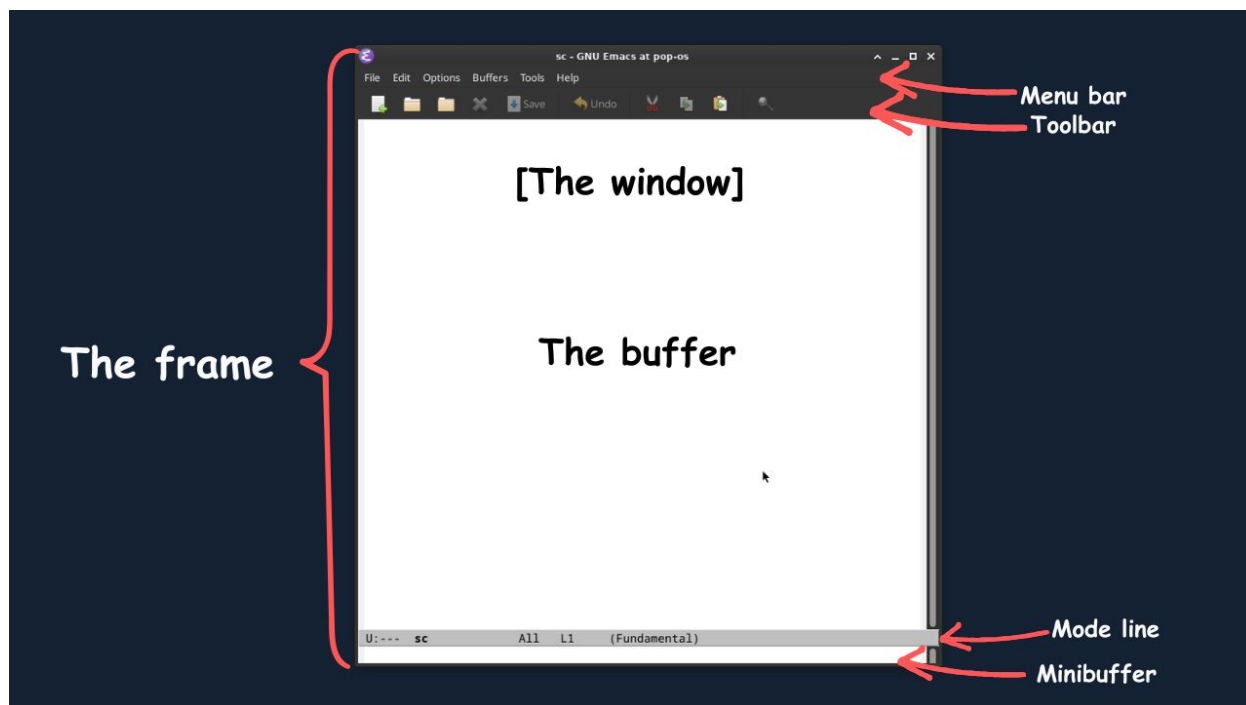
Emacs is a free, open source, extensible text editor. But it's so much more than that. Think of Emacs as a live environment for manipulating text on your screen. From the depths of my writer brain, I could not think up a better way to describe it.

Do you remember the scene in the *Minority Report* movie (<https://youtu.be/33Raqx9sFbo?si=YVq5nfTVVQyGZIjs>) when Tom Cruise is using his hands to slide, rotate, and zoom in on the holographic images on his computer? That's what Emacs feels like, but for your writing experience. Compared with the clunky experience of formatting your docs on Word, Emacs is like a dream.

But enough of the purple prose. I'm sure you're anxious to dive in.

Start by learning about the anatomy of an Emacs frame.

3 The Frame



The frame encompasses what is normally referred to as a window in most desktop environments you are probably familiar with. The Emacs frame is the floating, interactive space for everything in Emacs, including the menu bar, toolbar, buffer window, mode line, and echo area.

3.1 The menu bar

Emacs takes commands based on text input or key commands; most Emacs users prefer to issue commands to the system this way. But, by default, you have the menu bar (which can be disabled in your configuration). The menu bar lets you issue commands via graphical menus.

3.2 The toolbar

Like many contemporary software applications, Emacs includes a rich, graphical toolbar with icons meant to represent the functions they perform.

If you want to better emulate the look and feel of other graphical text editors you are familiar with, you may consider keeping the toolbar and menu bar configured as is; if not, you can see how to turn them off when we get to the section on configuring your interface.

3.3 The window

Here's where the fun begins. The Emacs window is the main playable area, the canvas upon which you will design, organize, and compose the work you do in Emacs. You'll find it just below the toolbar (if your toolbar is enabled). The window also contains your buffer space.

3.4 Mode line

The Emacs mode line displays important information about your current buffer. As the name suggests, it will give you information about what major or minor modes you are using. It can also be configured to give you all kinds of other information that may help you in your daily workflow.

3.5 Echo area (mini buffer)

The echo area, also known as the mini buffer, "echoes" text to you for various purposes. For example, if you run a function, or save a file, or get an error, the echo area will display text back to you about what's happening in Emacs. You can think of it as a status buffer. But, like everything else in Emacs, it's much more than that, and can display both small bits of text, or large lists of information.

4 Emacs keys and keybindings

In this chapter, we're going to grapple with the basics of Emacs keybindings. After reading this you should have a better understanding of how these keybindings can speed up your workflow and help you use Emacs more efficiently.

In a text editor, particularly with a minimalist interface such as Emacs has, you will rely more on your keyboard than the mouse. In fact, this is the way most Emacs users prefer to work—with finger-twitch speed and accuracy. We shall not shun the mouse entirely, but you will be using it a whole lot less in Emacs than in other Word Processors.

Thus, instead of clicking and dragging, you will learn to rely on speedy keybindings—like keyboard shortcuts but Emacs shortcuts, basically—to perform important functions for text manipulation.

You may find that this approach is faster, more efficient, and works more naturally with how you think; this is why I often say Emacs is an integrated development environment (IDE) for your brain.

(If you've already been using keyboard shortcuts like `Ctrl-c` and `Ctrl-v` for copying and pasting, then you're ahead of the game.)

Tip: At this point, I will pause and advise you to check out the built-in Emacs tutorial that shipped with the program. You can access this by pressing `Ctrl-h` and then the `"t"` key on your keyboard. This will launch the Emacs visual tutorial. It does a great job of introducing you to the basic key commands for Emacs.

For the sake of completeness, I will continue with my introduction to the keys and keybindings, but the built-in tutorial gives you everything you need to get started, and I suggest you work through that first.

4.1 Control, meta, and more

In the world of Emacs, the keys on your keyboard correspond with certain characters in the Emacs language and documentation.

C	the control key, corresponds to the <code>Ctrl</code> key on your keyboard.
M	the meta key, corresponds to the <code>Alt</code> key on your keyboard.
S	the shift key, corresponds to the <code>Shift</code> key on your keyboard.
s	(note the lower case) the super or "Windows" key, corresponds to the Windows or Mac key on your keyboard.

Except for the **M**, you probably could have guessed the others.

4.2 Keybindings and how they work

The Emacs developers have already gone to the trouble of creating a map of keybindings that run certain important functions. Basically, this is the global keymap of Emacs, and it is vast.

As you can imagine, you have the power to change or completely rewrite the map as you see fit. Though, as a beginner, you may want to stick with the default keys for now. But

don't let that discourage you. There are a lot of empty key combinations available for you to configure as you like without overwriting an existing one.

As you can further imagine, many of the programs and functionality built into Emacs, or packages you will install along the way, have already rewritten much of the original keymap. This is fine, and it's to be expected.

Here are some of the basic keybindings that will (most likely) never be overwritten by any other packages:

C-x C-s Save buffer to file.

C-x C-f Find file.

Notice, these keybindings involve pressing down the **Ctrl** key on your keyboard and pressing another key, but you don't have to remove your finger from the **Ctrl** key in order to punch the other characters. For example, to save the buffer, you can hold the **Ctrl** key down and then type an "x" followed by an "s." That's it. No sweat.

As you peruse through the Emacs documentation (with **C-h r**) you will see many other key commands just like these. Now you will know what they mean.

Let's keep going. I'm having some fun here, and I hope you are too. In the next chapter, we're going to tackle some bigger issues.

5 Navigation (moving around in Emacs)

Your first task in mastering Emacs is to just move around. That means opening files, yes. It also means moving around within those files. The navigational features of Emacs (and the various extended functionality available through installable packages) are extensive.

We're going to start with the basics to get you up and running. And I will later suggest ways to augment your experience. You also have the vast ecosystem of Emacs YouTube videos from yours truly (<https://www.youtube.com/@ChristopherMaiorana>) and other talented individuals out there. So let's just start with the basics.

5.1 Opening files in dired, the directory editor

"Dired" (pronounced "der," as in *derp*, "ed" as in Edward), the "directory editor," is a powerful file management system in Emacs. Like virtually everything else in Emacs, it's like a self-contained programming environment within the living environment of the Emacs editor.

With Dired, you can search, view, and open files and directories. For our purposes, that's just about the extent of what you'll be using Dired for.

To enter Dired in your current buffer, there's a simple key command: **C-x d**.

To open a file in Dired, there's another simple key command: **C-x C-f**.

5.2 Navigating the buffer

Now for the exciting part: moving around the text buffer.

Your default arrow keys on your keyboard will work just fine. You can use the up and down arrows to move up and down a text buffer line by line. This is fine. And you can use the left and right arrows to go left and right one character at a time. This is also fine. But it's nothing fancy.

Emacs is much more interesting than that. Start using these new commands, and repeat until they're second nature (and they will be soon enough):

- **C-n**: next line, or move down one line.
- **C-p**: previous line, or move up one line.
- **C-f**: forward character.
- **C-b**: back character.

That's good for starters. The keys above will do what your arrow keys do, but allow you to stay closer to your keyboard's home row. Already you're getting faster. But there's more. Emacs lets you move forward and backward by word too:

- **M-f**: forward word.
- **M-b**: back word.

I like moving forward by word while editing. But even so, that's a lot of button-pressing. We can do even better. Emacs lets you move forward by sentence as well.

- **M-a**: go to beginning of sentence.
- **M-e**: go to end of sentence.

Pretty cool, huh? This will be enough to get you started with file and buffer navigation. And you will learn more over time.

But what is a sentence? **Important to note:** by default, Emacs requires a sentence have two spaces after it in order to be considered a sentence. This will make sure your sentence-based functionality will work properly.

5.3 A special note on keybindings

For new users, the default Emacs keybindings may seem a bit cumbersome. You may be tempted to try out an alternate keymapping, like the popular Evil mode (which ports over Vim keybindings). I would advise avoiding this temptation for now. It is proper and good and right to learn the default Emacs keybindings, even if you are coming over from Vim. Heed my words.

6 Introducing Org Mode

Org Mode is a complete productivity suite integrated into Emacs. Remember when I said in the introduction to this little handbook that Emacs is a living environment for manipulating text? Org Mode is a prime example of that. The same text buffers you use to write your text are used for storing and retrieving organizational information from across a series of files.

Imagine having Emacs parse through a series of files for you and give you relevant details about projects, tasks, schedules, deadlines, and calendars.

The full capabilities of Org Mode are far outside the scope of this handbook, but rest assured, you don't need to know everything about Org Mode to start using it in your writing process. There are a few Org Mode functions we will take specific interest in. And that's all we need for now. These are: the outliner, the syntax, and the exporter.

6.1 The outliner

The argument for outlining versus not outlining continues to rage for writers.

- "Is it better to outline first?"
- "Is it better to just write and organize it later?"

Org Mode gives you the best of both worlds. You can write, organize, outline, re-arrange, all at the same time. Or you can do one, and not the others. Or, you can just write. It's your choice.

Org Mode uses headings to organize your document. Headings can also be nested under other headings, just like a hierarchical outline. (That's exactly what it is.) Headings are denoted by asterisks. (And, if you don't like asterisks, that can be changed.)

```
* Heading level 1
** Heading level 2
*** Heading level 3
```

And so forth. Of course, make sure you always nest child headings under a parent heading (obviously). Don't put a level two or three out there on its own.

For a demonstration of the outliner in action check out my video here (<https://youtu.be/6yRhWG28-84?si=9kSU32sSBHYkvT4d>).

6.2 Org Mode syntax

Org Mode offers the critical formatting options you will need to make your text stand out when necessary. But instead of highlighting and re-formatting text via the interface, you use specific characters to denote changes in style.

For italics, you can use forward slashes like this: I want to */italicize/* that word. In this case, the word "italicized" will be italicized.

For bold, you can use asterisks. I want to make the word ***bold*** stand out. The word "bold" will thus be bolded.

You can also do underlining with underscores. Let's get this _underlined_.

6.3 The Org Mode exporter

The world doesn't want your Org Mode file. In virtually every case, you will want to do some textual alchemy to convert your Org Mode document into some other deliverable format. This is fine. Org Mode can handle that.

You can export to HTML, PDF, Open Office, Microsoft Word, Rich Text Format (RTF), Markdown, LaTeX, and more. You can even export an Org Mode document to Org Mode (that's right).

We will go into more depth on the various export options at a later time (see Chapter 12 [exporting], page 21). For now, you can see your onboard export options by pressing **C-c C-e**. This will bring up the "export dispatcher." Remember the options you see here are only the ones that have been pre-selected to appear here. More can be added to this list, and some can be taken away.

Side note: you can browse different export "backends" by customizing the variable `org-export-backends`.

M-x customize-variable <RET> org-export-backends

Use the checkbox to select the ones you want and click "Apply and save." These changes are saved in your local Emacs configuration folder located in the home directory under `~/.emacs.d` on most installations.

6.4 The tag system

You will often find it helpful, or even necessary, to give additional context to headings in your Org Mode document.

This is why the tag feature exists. Not only can tags be used to give context to headings within your document, you can also use this as a sorting criteria in the document itself, or in your agenda view. (The Org Mode agenda is outside of the scope of this handbook, but if you'd like some more information you can check out my video (<https://youtu.be/HUNRMIOU7aw?si=GbgLfdFyovKHUFTS>).)

For now, just know you can add tags to an Org Mode heading by pressing **C-c C-q**. You will be prompted for a tag in your mini buffer.

In the example below, I have two headings, one tagged "rewrite" and the other tagged "noexport." The "noexport" is a special tag that will exclude that entire section from the final export; so, tags are not only helpful for context and sorting but also for giving special instructions.

```
* Spider pit sequence           :rewrite:
* Original ending               :noexport:
```

Well done. I am impressed at how well you are progressing through this journey. Take a rest, maybe get some coffee, and the book will still be here when you get back.

7 More syntactic goodness

Whether you're a writer (or just a thinker) you will want to have a solid understanding of how you can use the static Org Mode syntax for crafting beautiful, richly formatted documents.

In this section, we're going to take a closer look at some common syntax objects you may need as a writer of rich content.

7.1 Bold, italics, preformatted text

You've already seen how you can make your text bold using asterisks like ***this***, italics with forward slashes, like */so/*, as well as underlining, and tildes for `~code~` text.

If you have a whole paragraph or more of preformatted text you can use an "example" block:

```
#+begin_example
Plain text here.
#+end_example
```

There is a lot more you can do with structured templates like the above "*example*" block, but that is outside the scope of what most writers will be doing, so it will be covered elsewhere.

7.2 Subs and supers

Notice above, how I used "*example*" within the paragraph text. By default, Org Mode uses the underscore and the up caret ("^") to denote subscripts and super scripts. If you need these, they are enabled already. But if you want to ignore that you can disable them by putting this at the top of your file:

```
#+OPTIONS: ^:nil
```

That's what I use.

Also, do you want fancy quotes or straight "dumb" ones?

This for fancy:

```
#+OPTIONS: ':t
```

This for straight:

```
#+OPTIONS: ':nil
```

7.3 Structured blocks (quotes and verses and code)

You've already seen the "*example*" structured block. Now, there are a few more that would be particularly relevant for writers.

Org Mode blockquotes provide a syntax for denoting multi-line quotations. Here's an example:

```
#+begin_quote
Miss Brooke had that kind of beauty which seems to be thrown into
relief by poor dress. Her hand and wrist were so finely formed that
she could wear sleeves not less bare of style than those in which the
```

Blessed Virgin appeared to Italian painters; and her profile as well as her stature and bearing seemed to gain the more dignity from her plain garments, which by the side of provincial fashion gave her the impressiveness of a fine quotation from the Bible,---or from one of our elder poets,---in a paragraph of to-day's newspaper.

--- George Eliot
#+end_quote

In the example above, I used line breaks to keep the example from running off the page. However, those line breaks would be ignored in your final export. What if you want to preserve those line breaks? That's what the verse block will do for you. This is ideal for reproducing bits of poetry:

#+begin_verse
 Say first---for Heaven hides nothing from thy view,
Nor the deep tract of Hell---say first what cause
Moved our grand parents, in that happy state,

--- John Milton, /Paradise Lost/
#+end_verse

8 Your first "config"

You have now arrived at that all-important step of defining your first Emacs configuration. By this point, you have been introduced to what Emacs is, how to get around, how to open and close files. It's time to get your hands dirty with a little custom coding. Don't worry, you'll only need to copy and paste for now.

When you first start up Emacs, the question is, "What's your setup?" Your personal Emacs setup file is read at startup every time Emacs loads. This is where you can load all of your customization, packages, and preferences. But how does Emacs know which file to check? We're going to learn about all of this below.

8.1 Emacs Lisp is your customization language

Emacs Lisp is a variant of the Common Lisp programming language used as the backbone of the Emacs ecosystem. Lisp is a very old language that does things its own way, using a lot of parentheses, and a wacky syntax. But don't worry, you don't need to be an Emacs Lisp expert to customize Emacs to your liking.

However, you'll find that learning a little Emacs Lisp will greatly benefit you down the road, even though it's outside the scope of this tutorial. If you've ever searched for a certain Emacs functionality online, you may have landed in an Emacs forum in which someone has suggested a workaround using—you guessed it—Emacs Lisp. There's no getting around it. At just about every turn in your Emacs journey, you will need to employ some Emacs Lisp here and there in order to get some work done. But that's OK.

That's enough background information for now. Let's move to getting this configuration installed.

8.2 How to install your Emacs configuration

To make your onboarding process a little easier, I have gone ahead and set up a starter config for you and hosted it publicly at github.com (<https://github.com/cryptstophers/darkstar/blob/master/config.org>).

You'll probably notice some funny things about this configuration. First of all, it's in an Org Mode file. That's right. Org Mode lets you store code inside of the source block snippet that can be run in other contexts.

It's good for you to see this, because many Emacs users manage and document their configuration inside Org Mode files and then compile, or "tangle," the code to different places.

If you want to see this demonstrated, you can tangle the code from this config file by downloading it, opening it, and running `C-c C-v t`. But before you do that, be aware that this will overwrite your existing `.emacs` file with the code from the `config.org` file. So maybe you don't want to do that yet.

As an alternative, you can copy and paste sections of the Org file into your own personal configuration to see what they do. It's your choice. You can check out my video (<https://youtu.be/fbiSFlykolg?si=Nxb7v0dztrHIJigJ>) to see this in action.

If you're curious about what's inside the config, you will notice that the paragraphs surrounding the code snippets, as well as comments inside the snippets explain what each section does. I also talk more about in the video above.

8.3 Common errors you might see on startup

GNU Emacs is a richly documented program with a lot of built-in complexity. When something fails, the program tries to give you as much information as possible to help you debug it. Here are a few common errors you might see if your Emacs fails on startup.

8.3.1 No such file or directory

This means you are trying to load or access a file that Emacs does not recognize or does not exist.

For example, I might have something like this in my config:

```
(load-file "~/Desktop/hp1.txt")
```

Everything about this function is correct. The `load-file` declaration loads the contents of a file into Emacs. The problem here is the file I tried to load does not exist in my system, so I get an error:

```
Cannot open load file: No such file or directory, /home/user/Desktop/hp1.txt
```

8.3.2 Void function: "Symbol's function definition is void"

If you are trying to run a function that has not been declared or is incorrect, void, formless, and useless, you might get an error like this:

```
Symbol's function definition is void: evil-mode
```

This error was generated because my configuration tried to load up Evil mode, but I don't have Evil mode installed (and nor should you). ;)

8.3.3 End of file during parsing

You will usually see this error if you have a missing parentheses somewhere in your config. You might have typed something wrong, or accidentally deleted a paren somewhere. (Believe me, I've done it all.)

8.3.4 Invalid read syntax

Like the above "End of file during parsing" error, you'll get an "Invalid read syntax" if you have an extra parentheses somewhere.

8.4 Troubleshooting your config file (if you happen to get errors)

It can be frustrating when an error in your config is causing Emacs to fail or load only partially.

If you are getting errors on startup, you can always launch Emacs from your terminal with the "debug-init" flag like so:

```
emacs --debug-init
```

This will open Emacs with the debugger active and give you more precise details about where it has encountered the error.

In cases like these, your normal functions should work just fine. You can use `visit-file` (`C-x C-f`) or `dired` to edit your config file directly.

For deeper troubleshooting you can also start Emacs without loading a config file. The easiest way to do that is to pop up a command prompt and run this:

```
emacs -q
```

On Mac and Linux that should get you going again, but without any special configuration.

Remember, earlier in the series, when I cautioned you against using Evil mode. This is why. If you had to start up Emacs with no config, and suddenly all of your keybindings were wiped out, you'll feel pretty lost pretty quick.

8.5 Don't overdo it with your config

Once you get into a config mindset, it's very easy to go overboard and start trying to configure every little detail of your interface. Believe me, I know from experience. Just look back through my YouTube videos and you'll see how radically my interface changed over the years.

But no matter how much I tweaked my setup, I threw most of it away and settled back into a minimalist config. It's your choice, it's your canvas. I'll just say this: **don't let your obsessive tweaking get in the way of doing meaningful work.**

If you find you've gone too far, just throw it all away and start over. Your next setup will be a lot simpler and more organized.

8.6 Going deep with Emacs Lisp

If you want to go deeper into Emacs Lisp, there's a great tutorial that ships with your Emacs installation. Just press `C-h i` to enter the Help system. Then press "M" and start "Emacs Lisp. . ." Let your wonderful Vertico expansion do some work for you. From there, select the *Emacs Lisp Intro*. You may enjoy this.

9 Installing packages

On your journey toward mastering Emacs, it's easy to get overwhelmed with all the built-in functionality. On top of that, you will likely see some advanced configurations out there that have incorporated outside functionality via package installs.

In this chapter, you're going to learn how to reach out to Emacs package archives and install packages from them.

9.1 Setting up the package archives

You will find some pre-configured package archives in the first code snippet of the config file from the previous chapter. We have set up MELPA, ELPA, and also the "non-gnu" package archive.

Notice the list of package archives and their respective URLs in quotes in the snippet below:

```
(require 'package)
(setq package-archives '(("melpa" . "https://melpa.org/packages/")
                        ("nongnu" . "https://elpa.nongnu.org/nongnu/")
                        ("elpa" . "https://elpa.gnu.org/packages/")))
(package-initialize)
(unless package-archive-contents
  (package-refresh-contents))
(unless (package-installed-p 'use-package)
  (package-install 'use-package))
(require 'use-package)
(setq use-package-always-ensure t)
```

9.2 The magic of "use-package"

As your configuration absorbs more packages over time, it can become a chore to remember exactly what you have installed. And if you switch to a different workstation, you will want a portable configuration you can take with you without missing a beat.

A package called "use-package" makes this easy. "Use-package" simplifies the process of installing and configuring your packages via a simple command in your configuration file. For example, let's say you want to sync your emacs config on your home laptop and work laptop; all you need to do is copy your config file and let "use-package" download and configure your installs.

The "darkstar" config I provided previously installs use-package as the first command. It first checks to see if you have "use-package" installed. If not, it will install it. That done, Emacs can proceed through the file and do the rest of the work.

From there, the first package we come to is "vertico." Here is the "use-package" declaration for it:

```
(use-package vertico
  :ensure t
  :init (vertico-mode))
```

This is doing a few sophisticating things:

1. Tell Emacs to "use" vertico, and install if necessary.
2. "Ensure" that the package is installed via "ensure."
3. Enable `vertico-mode` on startup via "init."

As you search through package documentation, you will find examples of `use-package` declarations, but the most basic kind—as seen above—will be sufficient in most cases.

9.3 Package refresh

It's a best practice to first download the latest package information from the archives before installing anything.

To do this, we just need to run an interactive function. (An interactive function is a function that can be invoked by pressing `M-x`.) We will invoke the function `package-refresh-contents` to download the latest package information:

`M-x package-refresh-contents`

(If you installed the "vertico" package, you should see the auto completion options for this and other functions.)

Emacs will reach out to your chosen repositories to grab new information. It shouldn't take more than a few seconds, depending on your Internet connection.

9.4 Install packages

Then you just need to run a command to install the package.

And, of course, when you `<RET>`, that's referring to the Return or Enter key on your keyboard.

`M-x package-install <RET> <package name> <RET>`

With auto-completion options, names of packages will be presented. Just start typing and your options will narrow down. Once you have found the desired package, press Enter again and the package will be installed.

Remember, just because is installed, Emacs will not always know what to do with it. Some packages need to be activated, as in the Vertico example above with `use-package`. Always check the package documentation to learn about how to use it.

10 The magic of abbrev mode

Emacs gives the user a deeper level of control and power over their writing output. So you can get more done faster. Often, it's not the big enhancements that give the greatest freedom, but the little ones, like text expansion as a small nicety that can greatly improve your speed and typing accuracy.

"Abbrev" is a minor mode in Emacs that takes an abbreviation and expands it into full text.

For example, I can't remember how to spell Zettelkasten, but I use the word often enough that I set a "Zt" as an *abbrev* for the full word. This means whenever I type "Zt," Emacs will expand that snippet to the full word, *Zettelkasten*.

You can also use abbrev mode to automatically catch common mistakes in your writing. For example, due to typing too fast, I often type "teh" instead of the good and proper "the." In effect, this means "teh" is the abbreviation, and "the" is its expansion.

Important. Note that abbrev mode, like other minor modes in Emacs, is not enabled by default. You will need to either turn it on in your session like so:

```
M-x abbrev-mode
```

Or enable it at startup via your config file. I have it hooked up like this in my config so that abbrev mode is enabled whenever I visit a text mode buffer, which includes Org Mode buffers:

```
(add-hook 'text-mode-hook (lambda ()
                            (visual-line-mode) ;; that wraps text in the buffer
                            (abbrev-mode) ;; enables abbrev mode
                            ))
```

Or, you can set it standing on its own line, like this:

```
(setq-default abbrev-mode t)
```

However you choose to enable it, just make sure it's set.

10.1 Create your own abbrev definitions

First, you will need to decide if the expansion you want to create should be saved in the global abbrev table (usable everywhere in Emacs) or into the local table based on your current mode (only available in the current mode, Org Mode, for example).

For most of your everyday writing needs, I'd suggest starting with the local mode because there may be instances in which you are doing something out of your normal context, like Emacs Lisp coding, for example, in which your customary abbreviations will be useless or, worse, a disruption. It's up to you.

10.1.1 Setting abbrevs with key commands

To add a global abbrev, you can use this command: `C-x a g`. This will grab the nearest word behind your cursor and use it as the expansion, prompting you to provide an abbrev. That might not be what you want. If you want to use an entire phrase as an expansion, like expand "ttyl" into "talk to you later," for example, you will highlight the whole phrase, "talk to you later," and then key in the command.

For a local abbrev, which takes your current mode as the argument, just substitute the "g" in the keybinding for an "l" (for local): `C-x a l`.

10.1.2 With Emacs Lisp statements

In this section, you'll have a chance to learn about evaluating Emacs Lisp expressions.

Setting a global abbrev with Emacs Lisp can be done as follows. Copy this text into an Org Mode buffer and press `C-c C-c` on it. This command will do some Org Mode magic and ask you if you want to evaluate the code block. Press "y" for yes.

```
#+begin_src elisp
(define-abbrev global-abbrev-table "ttyl" "talk to you later")
#+end_src
```

Likewise, you could set an abbreviation just for text modes, which would include Org Mode as follows.

```
#+begin_src elisp
(define-abbrev text-mode-abbrev-table "ttyl" "talk to you later")
#+end_src
```

Once you evaluate this statement, the abbrev will be available right away.

10.2 Saving your abbrevs into memory

As I mentioned in the very first section, Emacs is a "living environment," so in order for the program to remember your abbrevs, Emacs will save them to a file that will read upon startup whenever you open the program.

By default, Emacs uses a file in your home directory called `.abbrev_defs`.

Upon closing Emacs, you will be prompted to save any new abbrevs permanently in your list of abbrev definitions. Answer yes to save it.

10.3 Unexpand an abbrev

There may be occasions in which you do not want an abbrev to expand. You can simply run the `unexpand-abbrev` function with `M-x` and you're golden.

11 Counting words and word counters

Part of your job as a writer is to track various relevant metrics. For writers, word counts are important; they are the key metric publishers and editors will need to know to make sure your content fits their specifications. Emacs can provide you with word counts and much much more.

You can easily count words using certain functions or packages and get a nice read-out in your modeline. However, I'll warn you that counting words dynamically can slow down Emacs in larger documents.

The easiest, most reliable way to get your word count is to use the `count-words` function that comes with Emacs. You can run it interactively with `M-x` typing `count-words`, or you can use the default keybinding: `M-=`.

11.1 Tracking lines and line numbers

Likewise, you can get line numbering in your buffer. Line numbering is more important in programming, but if you're writing poetry, or for any reason wish to see your lines counted, you can enable this feature.

Here is a line you can add to your config file:

```
(display-line-numbers-mode 1)
```

11.2 Relative line numbers

Relative line numbers is a wonderful little feature, but like dynamic word count packages, it can slow you down in a larger file.

Instead of displaying lines in a sequential fashion, from top to bottom, this will count your lines up and down starting at your current line.

Check out my blog post (<https://chrismaiorana.com/relative-line-numbers-in-emacs/>) on this to see it in action.

Here is how to configure relative line numbering:

```
(setq display-line-numbers-type 'relative)
(display-line-numbers-mode 1)
```


12 Export

Now is the moment you've been waiting for. In this chapter, you will learn how you can send your Org Mode documents out into the world as market-acceptable manuscripts. Which, in effect, means you have graduated from Emacs For Writers boot camp, and you're ready to write the great American novel, short story, or whatever it is you like to write.

Most markets do not accept plain text files, let alone Org Mode files—unfortunately. So this is a necessary milestone in your journey.

Important note: In order to follow this guide, you will need to have LibreOffice Writer (<https://www.libreoffice.org/>) installed on your computer. This is a free, open source alternative to Microsoft Word.

12.1 Some theory - filename separation

It's important to keep your files organized. You will have many files, but when it comes to writing there are two critical file types at the top of the heap: your origin files (a plain text Org Mode file, for example) and the deliverable files (a PDF or Word document, for example).

If you are coming over from the world of conventional Word Processors you are probably accustomed to your file and deliverable being all-in-one. **This is not the case now.**

There are many different ways to organize this workflow, as you can imagine. Because you will always be working from the Org Mode file, and exporting new documents when needed, the exported files are incidental. You can declare a destination file path for your exported documents by placing the "export file name" keyword in your Org Mode file:

```
#+EXPORT_FILE_NAME: ~/path/to/document
```

You may be wondering, why does that matter? By default, Org Mode will export your new document to the same directory as the originating document. To avoid clutter, I have all my Org Mode documents in one folder, and I send exports to a separate "export" directory.

This way, my Org Mode files are in one directory with other .org files and not cluttered up with PDFs, Word documents, and any other extraneous deliverable formats I may use at any given time. This helps keep things organized.

12.2 Using the export dispatcher

To engage the Org Mode exporter, you can use the existing keybinding: **C-c C-e**.

A bunch of export options will appear in a new buffer. Each export option is mapped to a key. The "Open Office" exporter, which we will be using, is mapped to the **o** key.

If you proceed to export your Org Mode document as is you will get an Open Office document that you can save as Microsoft Word. However, the default export styles will not be in standard manuscript format.

You will need to give Org Mode instructions on how you want the document formatted by using a template. The easiest way to do this is to edit an Open Office document exported from Org Mode to match standard manuscript format and save it out of LibreOffice as a template file: **my-template.ott**, for example. (You can name it whatever you want.)

Then, you will just need to tell Org Mode to use that template by putting this line in your Org Mode file:

```
#+ODT_STYLES_FILE: "/path/to/my-template.ott"
```

Now, when you open the Org Mode dispatcher again with `C-c C-e`, you can choose Open Office as the export vehicle, and your styles will be used.

I demonstrated this whole process in a video (https://youtu.be/WXYdqEFD_Fw?si=prJtEnUum7ag_Lw4).

12.3 Using my custom function

If you are feeling adventurous, you could add my custom function (<https://chrismaiorana.com/from-emacs-to-microsoft-word/>) into your configuration. All you need to do is update the titles and file paths.

This function (run independently from the export dispatcher) will prompt you for different hard-coded templates for export. This is helpful if you have different templates for different purposes and don't want to have to update the styles file keyword each time you export.

12.4 Exporting your documents like a pro

In summary, you should now have a clearer idea of how you can take an Org Mode file and export it out to a variety of different formats accepted by the writer's market.

Remember that different editors and publishers may have their own unique standards. Always be sure to read submission guidelines closely to make sure you format your document correctly. With Emacs and LibreOffice you have everything you would need to meet the specifications of the marketplace.

Now, all you have to do is come up with great ideas and write. And that's the hard part.

13 Research and note-taking

The following section is all about research and taking notes. I use the word "research" inclusively to mean the actual process of gathering research materials for writing but also the more passive process of capturing ideas as they come to you.

If you want to get into the nitty gritty of Emacs file organization feel free to skip ahead to the next chapter.

13.1 Value of notes

To my readers, it may go without saying, but: notes help you **remember everything**. Notes are way to get that much-vaunted "second brain" you hear so much about. It's just nice to have a place for all those stray ideas and thoughts so that you might use them later.

I keep notes for things like:

- Quotes from books I like.
- Lists of common names and surnames for fictional characters.
- Lists of uncommon words that have a special quality.
- "Vibechecks": things you feel like writing as a test but may never become something.
- Ideas that need some time to sit.

And much more.

13.2 Fishing for ideas

Ideas are everywhere. They will catch you by surprise. It doesn't matter if you're a fiction writer, blogger, or academic. You'll have great ideas appear to you as if from nowhere.

Maybe they come from God, or maybe from some deep place in your subconscious. It's like magic—and if you don't think so you're a muggle.

It doesn't really matter where the ideas come from. The point is: if you're mind is open, good ideas, like fish swimming by on a lazy river, will always be there. You just need to reach in and grab one.

Most important: your first job is to put those ideas somewhere where you can visit them often.

In this guide, I have recommended plain text—with Emacs—as the best storage and retrieval system, for reasons that may seem obvious to you: text is cross-platform, easy to back up, easy to search, and just simple.

In this section, I will propose some theory behind note-taking, and how you can do it in a fun, straightforward way—all the while *enjoying* the process. Then I will introduce you to my personal note system, with code, if you choose to try it out.

13.3 Make everything a note

This is where I will insert an opinion: I think you should make everything a note.

Hear me out.

Pick a directory. Call it "notes," "files," "text," whatever you want. Start putting plain text Org Mode files in there. You can organize them later. Just start putting everything in there.

It might be any of the following:

- A business idea
- A note about something you read
- An outline for a novel
- Really anything

13.3.1 A flat hierarchy

Alright, tricky word play here. There's no such thing as a flat hierarchy. But this is how I think of the "everything is a note" methodology.

With all of your notes in one directory, no file gets special treatment. They all land in the same place. There's no "Level 1" or "IMPORTANT" directory reserved for certain high-priority files.

I've found this to be a superior approach to having multiple directories based on themes, and even superior to having tags on files.

The point is to reduce cognitive load during the input phase. When you have a hot idea you just want to get it into the system with as little friction as possible. If you have to stop and think, "OK, what folder should I put this in?" or "How should I tag this?" you're already coming up against friction.

You run the risk of stopping dead in your tracks and re-thinking your whole tag system or file hierarchy.

The approach I recommend for notes is what I'd call a **content-first approach**. The contents of the file determine what the file is about rather than how it's tagged or what folder it's sitting in.

13.3.2 Simplicity

What I like about the "everything is a note" approach is that it cuts down on unnecessary decision-making.

If you're like me, you probably tend to overthink—especially in systems related to how you like to work.

Overthinking and over-optimizing can be a clever guise for **procrastination**.

13.3.3 Necessary tools

For a simple, file-based note system all you need is already given to you in Emacs.

- Emacs Lisp provides the functionality necessary to write new files into your system.
- Org Mode provides tools for hyperlinking related files.
- Installable tools like **consult-grep** paired with auto completion packages give you robust search functionality in case you lose track of a file.

Since implementing my own system I have not lost track of a single idea.

(Unless I neglected to put it into the system to begin with.)

13.4 "Binding" notes

As I've said, Org Mode provides all the tools you will need to get started on binding your notes together.

And we are taking a content-first approach, meaning the contents of the file are the sole-governing factors at play on that file's destiny.

So you may find yourself with a note containing some golden prose, and you may have other notes that are simply organizational notes that link to other notes.

I call those organizational notes "binder" files or "notebooks." You can call it whatever you want.

By using Org Mode hyperlinks you can connect your binder note to any related note.

13.4.1 Use cases

Good use cases for notebook binders include notes related to a single work, book, or topic. You may also have a binder note that outlines the chapters of a book or the sections of a short story.

Lots of people have an affinity for the Feynman technique, named after the famous physicist who kept notebooks as a way of learning difficult concepts. He would, in essence, explain the concept to himself as if he was teaching it to someone, and this would help solidify the concept.

You could do the same thing with your notebook or invent your own system that works for you and only you. It's your choice.

13.4.2 How to use links

As mentioned above, Org Mode provides a hyperlinking feature you can use to jump around your files. Just think of how deeply nested your information can be.

To create a hyperlink you just need to press **C-c C-l**. This will bring up a menu. If you want to create a link to a file just choose "file:" and you will then be prompted for the file path. This is another instance in which auto completion will be a life saver.

Then you will be prompted for the link text—this is the clickable part of the link.

That's it!

13.4.3 Hot tip: saving link patterns

You can create your own link templates to save some time linking to commonly linked items.

```
(setq org-link-abbrev-alist
  '(("duckduckgo" . "https://duckduckgo.com/?q=%s")
    ("cmweb-emacs" . "https://chrismaiorana.com/category/emacs/")))
```

A few things are happening in these examples. The first link in this list, the one that goes to the DuckDuckGo search engine is an example of a link that takes the link text as input. So you can, in effect, pass a value to the link. That's a nice feature to know about, but if you want to keep it simple you can simply provide a key and the link value.

What does this do? With these list items added to your **org-link-abbrev-list** the key portion will be an available option when you hit the **C-c C-l** function. In this example, my **cmweb-emacs** link will pass in the URL of my website's Emacs category. So I don't have to

copy and paste it or type it in manually every time I want to link back to my website. This saves some time, but most importantly avoids errors.

13.5 My personal note system

There are a lot of great note-taking packages out there for Emacs. I recommend trying all of them.

One of the pitfalls with taking on a note-taking package is that you have to learn the program itself while coming up with your own organizational system. You have the tools in front of you and have to figure out how to use them.

Instead, I found it best to build my own functionality that did only what I needed it to do:

- Create new files with a timestamp as unique id.
- Create a linked note from highlighted text (like in Obsidian).
- Turn a snapshot into a note.
- Rewrite another file in the system and save it in my notes directory.

All of this can be done manually, so I didn't need any extra functionality, but having the new functions basically makes it a breeze to use.

If you would like to use my note system, you can follow along with the rest of this chapter or skip ahead.

13.5.1 Unique files

Taking inspiration from the Denote system (<https://github.com/protesilaos/denote>), I wanted to use a flat file system with no database. I also wanted to make sure each file had a unique ID; this was accomplished by giving each new file a timestamp in the filename, then prompting the user for a keyword.

You can simply run the `nb/create-new-note` function with `C-c n`. You will be prompted for a keyword, then the file will be saved in the `~/Documents/notes` directory. I would advise updating the code with whatever directory you wish to use by replacing the quoted directory path for `base-dir`.

```
(base-dir "~/Documents/notes/") ; Specify the base directory for notes
```

13.5.2 Creating linked files from text

To automate the process of creating a new note and linking to the new note from your active buffer, I have a function that will take a region of highlighted text and use it as a link.

This is accomplished with the `nb/new-linked-note` function mapped to `C-c l`.

Simply highlight the text you want to link from, run the command, and create the note. The link will be created, but will *not* automatically jump you to the new note.

This is also a great way to leave a reminder to yourself that there is some additional information here you may want to expand upon, now or later.

13.5.3 Turn snapshot into a note

This part may not work just right on your system without some tweaking, but I wanted a way to be able to prompt myself for a new created via screenshot. This was accomplished with a bash script that uses **zenity** for a graphical keyword prompt and then opens up my screenshot app to take the picture, ultimately saving the new snapshot in my notes directory use the same filename scheme as all the other notes.

Of course, you don't want to have to open a terminal every time you want to take a screenshot. So I mapped this script to a key command in my desktop environment (which is XFCE).

You'll notice all files in the base notes directory, regardless of file type, uphold the same schema. There is no hierarchy. This goes along nicely with the Zettelkasten methodology.

(There, I found a way of working Zettelkasten into this handbook.)

13.5.4 Rewriting another file

There may be occasions in which I want to copy a file, as is, into the notes directory, and give it the proper filename scheme.

In my package, this is accomplished with a function called **nb/rewrite-note-file**. I did not map this to a keybinding, because I don't use it as frequently as the other functions, but you are welcome to do so.

This function will prompt you in Dired for the file you want to copy, then prompt for a new keyword. Then the file will be placed in the notes directory, as usual.

13.5.5 Downloading and using NB

As I said, I think it's best to come up with your own system. But if you want to give the NB system a try you can find the GitHub page (<https://github.com/cryptstopher/note-well>), download or copy and paste the functions, run them, and use them.

Of course, I provide no guarantee it will work well on your system, but if you like it let me know.

14 File organization theory

One of the major challenges writers face is squeezing in time to work on passion projects between day jobs and other responsibilities. Keeping files organized can save a lot of mental friction when you have but a few precious moments between one thing and another to get some writing done.

Conversely, if writing is your full-time occupation then having ready, simple, and functional, systems for your files can make all the difference between relaxed productivity and frustrated procrastination.

In this chapter, we will take a look at a few principles you can bring onboard, or ignore, as well as some Emacs functionality that can help you keep things organized.

When I talk about keeping things organized, I'm basically talking about files. Whether you are in the research, drafting, or editing phase—or coming and going between all three—it all comes down to files.

14.1 File temperature

As I mentioned in the chapter on research and note-taking, I primarily work out of one directory with no sub-directories.

I often refer to "file temperature" as a way to differentiate between files that go "out" to the public and files that are for my own purposes. Files that go out are "hot" and internal-use only files are "cold". Pretty simple.

So far, I've never mixed up files or sent the wrong one to the right place.

This is because Org Mode lets you send file contents to different destinations in a variety of ways.

You can block out headings from the exporter using the "noexport" tag. You can use comments to leave notes for yourself in a document. You can also set an "export file name" option, which will send your exported document to a destination of your choosing (that is, to somewhere outside of the working directory).

We will discuss these options later in the chapter.

For now, I introduce the concept of file temperature to get you thinking, as you create files for different purposes, which ones will be consumed by the public and which are for internal purposes, and how you differentiate between them.

14.2 File attachments

In the research chapter, I showed you how you can organize all files in one directory for easier management. In this section, we'll look at an opposite arrangement: organizing files under document headings.

That's right. You have already learned how to use headings to outline an Org Mode document. You can also use headings to create linked directories and save files there. These are Org Mode attachments. Like email attachments, but much cooler.

There are just a few simple steps. First, use the attach command (`C-c C-a`) to open the attachment dispatcher.

Then choose what you want to do. You can create a new file in the attached directory, open the directory, copy a file, and much more.

14.2.1 What's happening on the backend

When you create an attachment, Emacs will assign an id to the heading that matches a directory. The id is added as property.

By default, the new directories are saved in a directory called "data", but, like all options in Emacs, this can be configured to anything you want.

In your config, you customize the directory thus:

```
(setq org-attach-id-dir "/path/to/dir")
```

As an alternative, you can set the directory value as a property. This is what Org Mode would do automatically using a hash:

```
:PROPERTIES:
:DIR: dir/
:END:
```

14.2.2 Create a new attached file

There are many attachment options in the dispatcher. You'll see that as soon as you park your point on an outline node and press **C-c C-a**. Most commonly, you will probably want to create a file, which you can do by pressing **n**.

This will prompt you for a filename.

14.2.3 Open the attachment directory in Dired

Likewise, if you don't want to create a file but simply see what's in the attachment directory, you can press the **f** key at the dispatcher.

- Lowercase **f** will open Dired in another window.
- Uppercase **F** will replace your current buffer with Dired.

14.2.4 Use cases

In essence, attachments are a way to associate files with a heading in another file.

This can be great for content planning. For example, let's say you list out a bunch of blog posts you want to write as outline nodes in an Org Mode file. With attachments, you can associate files with those headings. The attached directories can hold research documents, ideas, or writing samples, virtually anything you want.

If you use Org Mode for customer relations management (which wouldn't be crazy at all), you could have each customer listed as a node in your file and use the attachments to keep work logs, communications, and any other files relevant to that customer.

Think about it.

14.3 Heading management

So far in our file organization theory section, we have covered file temperature, file attachments, and some related concepts. These tricks have involved managing a series of related files. Now, we'll take a look at segmenting and organizing contents within a single file.

There will often be occasions in which you will want to keep sections of your document out of the final draft. For example, you might have one section for outlining and another

section for drafting. Or, you might have a heading under which you've kept some personal notes about the file. Naturally, you won't want your reader seeing that.

Likewise, you may have a richly outlined document with multiple heading levels but want only paragraph text and no headings in the final draft. Or, maybe you want only level 1 headings included?

In both situations you have some convenient tags you can add to a heading to get the Org Mode exporter to do what you want it to do!

14.3.1 "Noexport" a whole heading

To totally exclude a heading from the final output, you can add a `noexport` tag. This will effectively tell the Org Mode exporter that this heading is basically invisible.

```
* Notes section                                :noexport:
```

This tag is inherited in each heading generation. This means if you "noexport" the level 1 heading, each subheading will be "noexported" as well. This is convenient because you don't have to put a tag on every heading. But, if you wanted to export a subheading from a "noexport" parent, it won't work. In that case, you will need to move that subheading out from under its parent node.

14.3.2 Ignore a heading

Ignoring a heading only excludes the heading text from the exporter. This means all content underneath will be included in your exported document.

Sadly, the ignore tag is not included in Org Mode by default. You have to install some additional packages to get this working.

Thankfully, I have an article on my website you can follow (<https://chrismaiorana.com/org-contrib/>), and a video.

Once you have the additional package installed, you can simply add an `ignore` tag to a heading to accomplish this awesome feat.

```
* Ignore this heading                            :ignore:
```

Once exported, just as expected, your heading text will be gone! Like magic.

15 Tracking goals and other metrics

In this section, we will do some work in the Org agenda, but not much. The agenda is a strong productivity tool, but its usefulness depends on your individual needs. Instead of prescribing a good way to use it I have highlighted a few points that will be most relevant to writers. Here we go.

15.1 Tracking deadlines with the agenda

As has been demonstrated on my YouTube channel and others, you can aggressively customize the Org agenda to do a lot of things. You can be as particular as you want, down to the last detail. Or, you can keep it simple.

For our purposes here we're just going to focus on assignment deadlines.

15.1.1 What is the agenda?

If you're not familiar with the Org agenda, I will briefly describe what it does.

The Org agenda builds a customizable dashboard for your tasks and projects and presents it an "agenda view" that you can customize.

You don't need any special programming knowledge to get a nice agenda view. The agenda uses Org Mode files you are already familiar with writing and builds a view with the heading data. Headings can be tasks or projects, with tags, categories, and other metadata.

15.1.2 Create a deadline

This is one of my favorite agenda features because it reduces some cognitive load. A deadline gives a project immediate priority, so you don't have to think too much about what you need to work on.

If you're familiar with the Getting Things Done (GTD) methodology, you likely already understand this. Calendar tasks are checked first due to their time sensitivity while "next actions" (unstuck from time commitments) can be dealt with as time allows.

The Org agenda gives us an easy way to do this. The deadline feature tells us how many days until an item comes due. And, the best part, you can set how many "warning days" you require to be properly aware of your task.

I have not found another open source program that will tell you how many days you have left until an item is due and conveniently place it in the same view as your other tasks.

To attach a deadline to a heading you can use the `C-c C-d` keybinding which is mapped to `org-deadline`.

A calendar window will pop up at the bottom of your buffer. Select the date or cycle forward until you have found the appropriate month.

15.1.3 Adding your file to the agenda

In order to see your deadlines in the Org agenda, you will need to add the file to the list of agenda files.

There is a key command that does this: `C-c [`. This runs the `org-agenda-file-to-front` function.

If you prefer to use Emacs Lisp in your config file:

```
(setq org-agenda-files '("/path/to/file1.org" "/path/to/file2.org"))
```

Now you can open the agenda view with **C-c a**. Press **a** again to see the basic agenda view, including your deadline items.

15.1.4 Warning days

As mentioned above, Org Mode lets you set a number of "warning days" for your deadline items. This is the number of days in which you want advance notice of your upcoming deadline. For example, if you only want to see your task in the agenda when it comes two weeks til the deadline, you can set your warning days to 14. This way, you won't even see the item in your agenda until the deadline is 14 days away.

In my case, any deadline within the next year is fair game, so I set my warning days to 365:

```
(setq org-deadline-warning-days 365)
```

Well done. You have the tools, you have the talent. Now, you just need to hit your deadlines.

15.2 Logging notes on headings

Org Mode has a nice log feature that lets you take notes on headings without interfering with the content beneath the heading. So if you want to make notes on the progress of a project as you go, this is something you will want to try.

The key command to add a note to a heading is **C-c C-z**, which runs the `org-add-note` function. If you run this function and save a note, Org Mode will add a timestamped note under your heading.

But of course, this note will appear in the main content area of your heading, and you may not want that.

To hide your note in an Org Mode "drawer" you will need to customize the variable, `org-log-into-drawer`.

You can also set and unset this value in your individual files by placing these startup declarations at the top:

```
#+STARTUP: logdrawer  
#+STARTUP: nologdrawer
```

Or, for your config, this bit of Emacs Lisp will work:

```
(setq org-log-into-drawer t)
```

Remember, because Org Mode drawers are folded by default when you open your file, you can save as many hundreds or thousands of notes as you like and have them neatly hidden from view.