Emacs

Why use Emacs for things like cloud computing even in the modern day?

Such applications are at the mercy of networking, which are orders of magnitude slower than operations on a local file system. Thus, they care a lot about **throughput** and **latency**, and terminal interfaces can deliver because they are simple and fast.

Implementation

Emacs is built in "layers": it has a C interpreter inside it, and atop it is a Lisp interpreter. The bulk of Emacs is written in Lisp.

```
+------------------+
| Lisp code        |
+------------------+
| Lisp interpreter |
+------------------+
| C interpreter    |
+------------------+
```

The I/O devices like mouse, keyboard, and display communicate with the application through the Lisp code.

Programs typically have their own interpreters embedded in their own executables. For example, Chromium executables come with a JavaScript interpreter included in them.

Concept of Buffers

- Emacs makes use of **buffers** to be *fast*. Buffers are just a bunch of text that live in RAM.
- Emacs (and editors in general) makes a clear distinction between what's *persistent* and what's not to balance speed and reliability. For work that is rapidly changing like when you're typing a sentence, we have very performant buffers. Only when work is ready to be saved, the application can flush the buffer to the file system in one fell swoop.

Buffer-related Key Binds:

```
C-x C-b                 list buffers
C-x b <buffer name>     switch to a buffer
```

Auto-generated Files

- By convention, file names starting with `.#` indicate a symbolic link to a file that is currently being edited by another program.
- When editing a file `F` in Emacs:
    - Emacs creates a symlink `.#F` that signals other programs that the file is being edited.

    - Emacs creates a file `#F#`, a copy of the unsaved buffer for `F` as part of its auto-save feature, a safety mechanism for in case Emacs or the computer crashes. This file disappears on write.

Editor Navigation: Moving the Point

NOTE: arrow keys (and their Shift and Ctrl variants) work too, but many of the movement keys are actually universal across Unix, which could be useful to know. Also, it's faster to use key binds that are in the main keyboard area instead of having to reach for arrow keys.

> Trust me, I spent multiple hours customizing and mastering my VS Code key binds over the last quarter. It's fun as FRICK.

The current cursor position is called the **point**.

Smaller strides visualized:

```
                        ^ C-p
                        |
C-a <----- M-b <-- C-b <-+-> C-f --> M-f -----> C-e
                        |
                        v C-n
```

At the **character level**:

```
C-f     Move point forward one character
C-b     Move point backwards one character
```

At the **word level**:

```
M-f     Move point forward one word
M-b     Move point backwards one word
```

At the **line level**:

```
C-e     Move point to end of current line
C-a     Move point to start of current line
C-p     Move point up one line
C-n     Move point down one line
```

These are especially useful to set up subsequent strokes. Common patterns include:

- `C-a C-k` to delete the entire line
- Starting new lines with `C-e RET` (below), `C-a C-p RET` (above)
- etc.

At the **sentence level**:

```
M-f     Move to end of next sentence
M-b     Move to beginning of previous sentence
```

At the **page level**:

```
C-v     Scroll down one page worth
M-v     Scroll up one page worth
```

At the **buffer level**:

```
M-<     Move point to start of buffer
M->     Move point to end of buffer
```

From **anywhere**:

```
M-g M-g NUM RET     Go to line number NUM
```

These are especially useful when setting up searches for the "first/last occurrence" of something.

At any point, you can use `C-l` (that's a lowercase L) to vertically center the point if possible. Use `C-l` again to bring the current line to the top of the viewport.

Fixing Mistakes

It's important to memorize these few to know how to get back to familiar territory when something goes wrong:

```
C-/             Undo the last command (only if text was modified)
C-g             Quit current command or exit command minibuffer
ESC ESC ESC     Universal "get out" - quit command, window, etc.
C-x C-s         Save current file
C-x C-f         Open/return to a file, creating it if necessary
C-x C-c         Exit Emacs
C-z             Temporarily suspend Emacs (enter fg to re-enter)
```

Help System

The help system makes Emacs a "self-documenting" system. `C-h` is the designated **help key**, which prefixes commands related to viewing documentation.

```
C-h b                           list key bindings
C-h k KEYSTROKE(S)              list one binding
C-h a <regex> RET               search for a command
M-x apropos RET <query> RET     search for a command
```

Extended Commands

`C-x` is the designated **eXtended command key**, which starts keystroke chords for common commands, like `C-x C-f` to open a file.

`M-x` is the designated **eXtended named command key**, which focuses a command **minibuffer** at the bottom of the terminal, below the mode bar. The minibuffer is itself a buffer supporting your familiar navigation keys, and in it you can write the full name of commands.

After attempting to use the full name for a command that has a `C-x` shortcut, Emacs will tell you what the shortcut is for future use.

There is a special command that allows you to send a numerical argument to a command:

```
C-u NUM KEY
```

For many commands, this simply repeats the actions, but some commands may have slightly different behavior. For example:

```
C-u 2 C-k       delete content of two lines and their newlines
C-k C-k         delete content of a line, and then its newline
```

This behavior is determined by how they're implemented - whether they take that optional numeric argument in the first place and what they do with it.

Window and Buffer Navigation

```
C-x 2       split current window into two
C-x o       switch focus to the other window
C-x 0       kill current window
C-x 1       kill all windows except the current window
```

Opening another window for the same buffer does not duplicate the buffer; any edits in one buffer will affect the other(s). You can still use this when you want to reference some other part of the buffer while typing in another region.

```
C-x C-b     list the current buffers
C-x b       switch to a specific buffer
```

```
C-x s       prompt save for some buffers
C-x 4 C-f   open file in another window
```

Shell within Emacs

Run a command as a separate, one-off shell process:

```
M-! <command> RET
```

Same thing but taking input from a buffer and then piping it to the shell command (takes all characters in **current region** and pipes them to the command as its stdin, and then takes the output of the command and pipes it to the *shell command output buffer* like normal):

```
M-| <command> RET
```

Selection Manipulation

Concept of the "current region (of current buffer)":

- You can save pointers called **marks** at arbitrary positions within a buffer.
- The **current region** is all the characters between the mark and the point.

You can set a mark at the current point with `C-SPC` or `C-@`. An example of selecting a region of text:

```
M-<             go to start of buffer
C-@             set marker
C-s eggert RET  search for "eggert"
M-|             pipe buffer into a shell command
```

You can find out where your mark is with `C-x C-x`, which exchanges point and mark (selects the text between them). You can `C-g` to cancel the selection.

Text Manipulation

Emacs distinguishes **killing** from **deleting**. When you **kill** text, it is actually saved in a special buffer called the **kill ring**, and content in here can be reinserted (aka **yanked**) at the point. Most commands that perform bulk removal of text actually *kill* the text, not *delete* it.

```
C-k     kill from point to end of line
C-w     kill current selection
M-w     copy curernt selection
C-y     yank most recent kill
M-y     cycle the text to yank through kill history
```

Modes

Emacs is a **modeful** editor. That means the current state of Emacs not only includes the contents of the current files being edited but also what way you intend to be using the editor next. A **mode** is like a method of interacting with the editor.

- **Upside:** more efficient for experts
- **Downside:** confusing/tricky for non-experts

```
M-x MODENAME            switch to mode
```

`C-h m` brings up a buffer that describes what mode you are in. The default "editor" mode is called **Fundamental** mode, and there is also **Text** mode in which navigating among words with certain characters like apostrophes is slightly different.

You can also open a shell subprocess (**Shell** mode) or a view of a directory (**dired**, directory-editing mode). The mode you are in affects the keys you input. You can see the name of the major mode you are currently in with the **mode bar** just above the minibuffer.

```
C-x d <dirname> RET     enter dired mode for directory
```