Keyboard Shortcuts every Command Line Hacker should know about GNU Readline

Most command line programs that offer line editing – like bash, Python, GDB, psql, sqlite and more – do so using GNU readline. Readline's a powerful library that grants history, completion, movement and editing to programs that use it — and a stable and consistent set of keyboard shortcuts.

Shame, then, that even serious command line

hackers never bother learning about its capabilities, as they can supercharge your command line productivity.

By *Mickey Petersen (/about)*

There's a large body of command line hackers who, despite their incredible skill and speed, never cottoned on to the fact that *line editing* – editing text on a prompt, like the one in Python or bash – is a skill that is transferable *and* worth learning.

Line editing is not a novel concept, nor a new one. On Linux, the most common *line reader* is GNU readline. It provides a framework for basic completion, movement, editing and prompt history.

In bash it's used when you press `<TAB>` to complete a file name; navigate through your input history with the arrow keys; or edit and type text at the prompt. In fact, it's so ubiquitous – yet veiled behind the programs it supports – that people credit *bash*, and not readline, for many of its novel features.

But GNU readline's a lot more capable than you might think. By default it uses Emacs key bindings, but you may already know that. Yet a surprising number of hackers – even ardent Emacs users – never bothered exploring its capabilities. (There is also, in the interest of full disclosure, a Vi emulation layer.)

Most people plateau when they learn about `Control+r` for reverse history isearch. But there are many reasons to learn keep learning:

- **Quick Keyboard Macro Recording**

  Tired of tediously manipulating a complex prompt when you decide you want to pass the prompt output to `fzf` or `grep`?

  Maybe you keep having to type the same repetitive incantation in `gdb` or `python`, over and over again?

  Maybe you have to write the same `SELECT` statements over and over again in `psql`?

  So why not use a keyboard macro? You can record, play back, and save them for later. And they can – if you want – work across programs that use GNU readline.

- **Smarter History Use**

There's more to GNU readline's history commands than merely navigating up or down the history list. Most are specialized; but they are all worth learning about.

- **Faster Movement and Editing**

  Learn a handful of Emacs key bindings and they'll pay dividends across a swathe of programs. Whether it's bash or SQLite's command line interface, the key bindings remain the same — and some even augment the defaults with specialized commands specific to that program.

  Memorize once, use (nearly) everywhere!

Of course, most Emacs hackers don't bother, because Emacs has its own shell integration (/article/running-shells-in-emacs-overview), and that does the job 99% of the time.

So why bother learning about GNU readline if you're an Emacs maven?

Because, when you next find yourself out in the cold – or if you wholly reject the warm embrace of Emacs – you'll have a set of standardized key bindings that greatly improves your productivity. There's also a rich set of features you can access with little effort, and they work well in a wide range of disparate command line applications.

So if you're sick and tired of reaching for the arrow keys just to edit or move around the prompt: Hark! For there are better ways abound.

Basics, Caveats & Useful Defaults

But, let's get the basics out of the way first.

GNU readline has its own little configuration language, and the global settings for it are usually found in `/etc/inputrc`. When it's done reading the global file, it'll check for a user-local file in `~/.inputrc`.

Like Emacs, readline respects Emacs's notation for shortcuts: `C-` is Control; `C-M-` is Control and Alt in unison. Like Emacs, readline is also capable of simple chords: `C-x p` and `C-x C-r` are both legitimate chords: press and release `Control+x` followed by `p`, for instance. So far, so Emacs.

GNU readline is not universally used. If you're not using Linux, your application may not use readline at all. And some programs have their own line editor implementation, like Zsh.

The `.inputrc` file

However, in the `.inputrc` file you should know that the `M-` qualifier does *not* work. Instead, write `\e` for Meta; and `\C` for Control. There is also a long-form format, but I find it cumbersome and the parser a bit wonky. And in any event, in a readline keyboard macro – more on keyboard macros later – you *have* to use this notation anyway, so you may as well standardize on the shorthand notation.

```
# Correct - binds to C-x q
"\C-xq": "Hello, World"
# Correct - binds to C-M-p
"\e\C-p": "Good Bye!"
```

Variables are set with `set`:

```
set expand-tilde on
```

You can also limit settings to certain programs:

```
$if Bash
   # ...
$else
   # ...
$endif
```

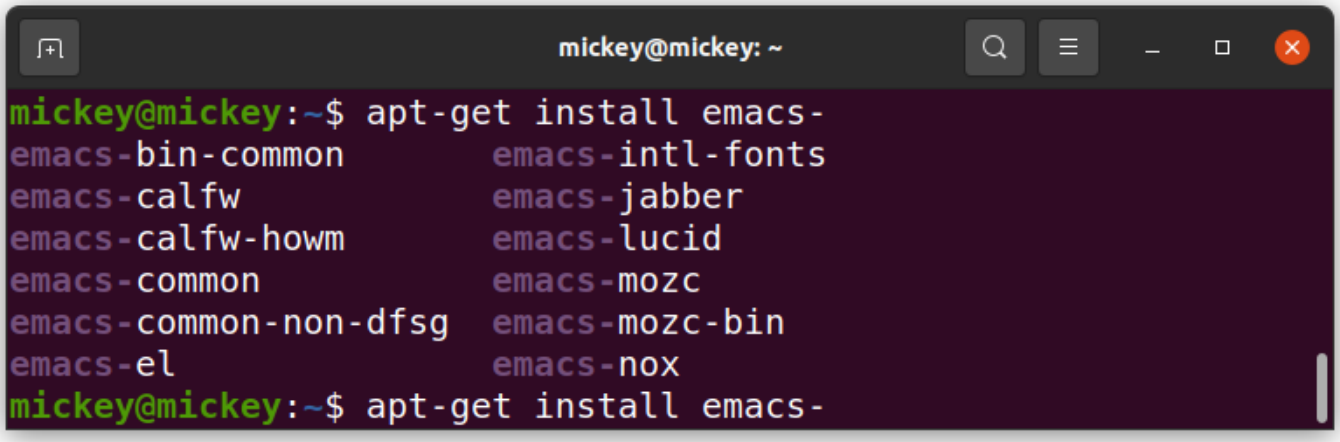
Determining the name is not easy though without reading the source.

Useful Defaults

One thing that catches people out about Emacs-style key bindings in readline is its hidden timeout. By default the `keyseq-timeout` value is a rather ambitious 500 ms. I find that far too quick, so I change it 1200 ms:

```
set keyseq-timeout 1200
```

Most distros seem to leave readline's color highlighting for completion disabled. I don't know why, as it's helpful.



Colored completion of partial matches in `bash`

```
set colored-stats on
set colored-completion-prefix on
```

Readline looks at your `LS_COLORS` environment variable to determine what the highlighting should look like. If it's missing, it'll use a reasonable default.

Basic Navigation and Editing

For command line editing, knowing a handful of basic keys is all you need to greatly speed up your typing and, on that account, your productivity. If you're already familiar with Emacs's movement keys (/article/effective-editing-movement), then you'll know them all, already.

Key Binding	Description
`C-b, C-f`	Move backward/forward one character
`M-b, M-f`	Move backward/forward by word
`C-a, C-e`	Move to the beginning/end of the line
`C-w, M-d`	Kill word backward/forward
`C-d, Backspace`	Delete char forward/backward `C-d` may send EOF on an empty line
`C-k, C-u`	Kill (to clipboard) to end of line/beginning of line
`C-y`	Yank from kill ring
`M-y` (after `C-y`)	Cycle through kill ring history
`C-t, M-t`	Transpose character/word
`M-u, M-l, M-c`	Upper, lower, or capitalize forward word
`C-_, C-x C-u, C-/`	Cycle through the undo list

Although GNU readline maintains a *kill ring* (clipboard), which works much like Emacs's kill ring, it's not shared with your system's clipboard, nor any other instances of GNU readline. It is, in effect, local to the process you're running.

GNU readline also has limited support for Emacs's universal and numeric arguments, so it's possible to type `M-2 M-d` to kill two words forward. You can also supply negative arguments with `M-- M-u` uppercases the word preceding point.

Undo is also worth knowing about. It's different than Emacs's undo ring, as it's linear, and is reset when the prompt is.

Knowing how to go to the beginning and end of a line; deleting and moving by word; and a little bit about killing and yanking will go a long way.

Taken together and you have the rudiments of Emacs's most basic editing suite available in a wide range of command line applications.

Less Useful Key Bindings

Key Bindings	**Description**
`C-@, M-SPC`	Set mark
`C-x C-x`	Swap point and mark
`C-q` or `C-v`	Quoted insert
`C-], M-C-]`	Character search forward/backward
`M-#`	Insert a comment
`C-x C-r`	Reload `.inputrc` file(s)
`C-c C-l`	Clear screen

The concept of the *mark* – one half of the dynamic duo, with point being the other one, that makes up the text selection region in Emacs – is also present in GNU readline. However, there is no support for visualizing the selected region: you're going to have to remember where the mark is! This is like Emacs before `M-x transient-mark-mode` became a thing.

That, combined with the editing scope limited to a line of text, means it's not really such a useful thing to know about.

Quoted insert *is* useful if you regularly have to insert keyboard escape codes. Like `^[` for ANSI control codes. To insert `^[`, type `C-q ESC`.

You can also comment out the line with `M-#` but I find its utility minimal in a line editor where commenting out a line has narrow applications.

Finally there's `C-x C-r`. That key is really only useful if you're experimenting with your `.inputrc` file. Invoking the key binding will silently force readline to reload its settings. That means you can experiment and not have to restart your program to test your changes.

Keyboard Macros

This is where, in my opinion, the value of GNU readline really kicks in. Like Emacs, readline has a keyboard macro recorder. It's nowhere near as advanced as Emacs's keyboard macro recorder (/article/keyboard-macros-are-misunderstood); but you only need the basics.

You can record, play back, and print out keyboard macros.

That is more useful than it seems, and it can save you a lot of typing:

1. If you're a frequent user of `fzf` – or ezf, if you want the Emacs equivalent (/article/fuzzy-finding-emacs-instead-of-fzf) – then I'm sure you have ritualized the practice of piping stuff to said tool:

```
$ apt-cache search emacs
```

Could be transformed into this with a quick'n'dirty keyboard macro:

```
$ $(apt-cache search emacs | ezf -f 1)
```

At that point you're free to take the command substitution and feed it into a command of your choice — like `apt-get` possibly. Remember, you can leave the point wherever you like: at the beginning or end of the prompt; or maybe at the pipe, so you can quickly change the parameters.

2. You can wrap prompts in simple commands, like `dir( ... )` or `help( ... )`, in Python.
3. Keyboard macros can be made global or local and written by hand. No recording required.
4. You can create one on-the-fly, and only later bind it to a key if you find it useful enough to keep around.

To truly make use of the macro functionality you must first bind the command `print-last-kbd-macro` to a key in your `.inputrc` file:

```
"\C-xP": print-last-kbd-macro
```

Here I bind it to `C-x P`. But feel free to use another key binding. Note the backslashed `C`.

Key Bindings	Description
`C-x (`, `C-x )`	Start/Stop macro recorder
`C-x e`	Play last macro
`C-x P`	Print out macro (Note you must bind the command to a key for this to work)

To start recording type `C-x (`. Unlike Emacs – or indeed unlike many other parts of GNU readline that signal state changes, like prefix arguments – it does not tell you it's recording, which is quite infuriating. Nevertheless, you can stop recording with `C-x )` and then play back your newly-minted macro with `C-x e`.

I would recommend you experiment a bit. There are few things it can't record. It's limited to *just* this line; you cannot macro record (or play back) across multiple prompt invocations. Keep that in mind.

Calling `C-x P` (or a key binding of your own choosing) spits out the macro string needed to replay the macro. You can put it in your `.inputrc` after binding it to a string, but **make sure you double quote it** if it is not quoted already.

Here's one that calls out to `fzf` (or `ezf`), bound to `C-x F`:

```
"\C-xF": "\C-e | fzf)\C-a(\C-a$\C-b\C-a"
```

If you look closer, you'll see that it's just a string with a bunch of key bindings. So you can indeed write your own.

To test it works, type `C-x C-r` to reload your `.inputrc`.

I personally think the keyboard macro functionality is an underrated gem. I can't imagine too many people know about or use it. Which is a real shame!

Text Completion

Whether it's `bash` or another tool of your choosing, it may have some form of TAB-based completion. By default the builtin GNU readline command `complete` is bound to `TAB` or `C-i` (they're one and the same control code, in actual fact.) But what most people *don't* know is that is just the beginning.

Bash, for instance, extends the completion system and adds a host of specialized completion commands.

Key Bindings	Description
`TAB, C-i`	Complete at point
`M-?`	List possible completions at point
`M-*`	Insert all completions
`M-~`	Complete tilde

The most common – the one everyone knows – is `TAB`. When a developer integrates GNU readline into their application, they may decide to support readline's underlying completion engine. Python does this; that's why you can do basic completion with `TAB` in a Python shell.

There are a couple more that are quite specialized, but worth knowing about.

`M-?` directs readline to display the completions at point. This is pretty much the same as hitting `TAB` multiple times, but with a crucial difference: `M-?` won't autocomplete parts of your query as `TAB` would.

`M-*` is more interesting. It takes all possible completion matches – sourced from `M-?` – and inserts them into your prompt. Now you can edit them before you pass them to another program.

If you frequently use `~` – referring to your home directory – you can use `M-~` to complete against it, as though you'd `TAB`-completed. Other programs may not implement it, or they handle it differently. Python inserts the path

to your home directory, for instance. The variable `expand-tilde` partly controls this.

History Search

Every GNU readline program implements its own unique history list, but (as you'd expect) they share a common set of key bindings you can use to navigate that list.

Key Bindings	Description
`M-<, M->`	Go to beginning/end of the history list
`C-r, C-s`	Reverse/Forward isearch history
`M-p, M-n`	Query search backward/forward through history list
`M-.`	Yank (insert) last argument
`C-p, C-n, Up, Down`	Previous/Next history entry

I'm sure you're familiar with the `<up>` and `<down>` arrows to move through history. `C-p` and `C-n` serve the same purpose.

Both `M-<` and `M->` move to the very beginning and end of the history list. Not *that* useful, but if regularly cycle through the history then it's useful to know that you can do so from either end, at will.

The history commands I love most, aside from navigating up or down, are the *isearch* – incremental search – history commands bound to `C-r` and `C-s`. Both search the history, incrementally, as you type. They're incredibly useful

and a hallmark of readline. Lots of people know about `C-r` but somehow never tweaked `C-s`: it searches forward, and you can mix and match the two to skim through the history matches.

> **NOTE**: If `C-s` freezes your terminal, you can blame the superannuated "flow control" feature — it stops the flow of text so you can pause at will and read it. Wonderful feature in the 1980s when your teletype would type faster than you could read the text. Less so today.
>
> Typing `stty -ixon` disables it.

In addition to incremental history search, there's `M-p` and `M-n`. Type either and you're asked to input a search string. Pressing `RET` jumps to the first match. It has its uses, but if you normally run your shells in Emacs (/article/running-shells-in-emacs-overview) you'll find it frustrating as `M-p` and `M-n` is how you navigate history in `M-x shell`.

Put this incantation in your `.inputrc` to rewire those two keys to also step through the history:

```
"\ep": previous-history
"\en": next-history
```

That leaves `M-.`. It inserts the last argument of the last command; repeat it and you'll step through previously-used prompt arguments. I find it useful if I regularly have to regurgitate a common argument:

```
$ grep *.txt foo.*
# Typing M-. now inserts foo.*:
$ rg ▉  # -> rg foo.*
```

Discovering Key Bindings

Because GNU readline is so extensible, it's not uncommon for programs to create their own commands and key bindings that apply only to that program. The key bindings I talked about earlier are, broadly speaking, available everywhere — even if they don't really make sense in the context of that application.

But bash, being a GNU product, has a large range of extended key bindings and commands you should know about.

But in order to discover them you really need to know about the *dump* commands. They are unbound by default. Here's what I have bound them to:

```
"\e\C-f": dump-functions
"\e\C-v": dump-variables
```

Typing `C-M-f` prints out all known functions; `C-M-v` all known variables. Give them a prefix argument (e.g., `M-1 C-M-f`) and readline spits out `.inputrc`-compatible code you can paste into your own settings file.

Bash-specific Key Bindings

There are many; so I encourage you to explore yourself. But I'll list a couple of my favorites.

Key Bindings	Description
`C-M-e`	Expand environment variable
`M-!`	Complete command
`M-/`	Complete filename
`M-@`	Complete hostname

`M-$`	Complete variable
`C-x C-e`	Edit and execute command in Emacs
`C-M-o`	Expand using *dynamic abbrev*. (Unbound by default; see below.)

`C-M-e` takes an environment variable at point and expands it inline. Useful for things like manipulating the order of things in `$PATH`, or tweaking the contents of one variable before storing it in another.

The other completers – `M-!`, `M-/`, `M-/@`, `M-$` – attempt a specialized, narrower version of the common completer bound to `TAB`. They have their uses: when you must insert a variable, for example.

Dynamic Abbrev (/article/text-expansion-hippie-expand) is an Emacs feature that has made the leap – sort of, anyway – into readline. It looks at the word at point and attempts to expand it using previously-seen history values. So, if you previously typed in a complex word – say a hostname – you can ask readline to recall it anywhere in a prompt:

```
$ ping secret.setec-astronomy.example.com
# Typing C-M-o expands to the full hostname.
$ ssh sec
```

It is not bound by default, so I recommend `C-M-o`:

```
"\e\C-o": dabbrev-expand
```

Repeated invocations cycle through the matches. It's a *fantastic* command.

Finally I want to mention `C-x C-e`, which is really quite cool. When you invoke `C-x C-e`, then bash/readline will send the current prompt to your `$EDITOR` for further processing.

If you have your Emacs running in client-server mode (`M-x server-start`, for instance) and your `$EDITOR` set to `emacsclient` then you can indeed edit the prompt in Emacs before returning it from whence it came with `C-x #`. Great for more complex edits.

My Settings

Here's a snapshot of my `.inputrc`. It's mostly about making things even more Emacs-like, plus a couple of basic macros.

```
### Mickey's .inputtrc

# Change the timeout for key sequences as 500ms is too fast.
set keyseq-timeout 1200

# By default, completions are not highlighted in color.
set colored-stats on
set colored-completion-prefix on

### BASH

$if Bash
# Wrap a command in $( ... )
"\C-xq": "\C-a\$(\C-e)"
# Wrap a command in $( .... | ezf -f 1)
"\C-xF": "\C-e | ezf -f 1)\C-a(\C-a$\C-b\C-a"

# C-M-o is dabbrev-expand
"\e\C-o": dabbrev-expand

$endif

### Python

$if Python

# Wrap prompt in !help( ... )  (for PDB)
"\C-xh": "\C-a!help(\C-e)"
# Wrap prompt in dir( ... )
"\C-xd": "\C-adir(\C-e)"

$endif

### Global
```

```
# Prints the last recorded macro
"\C-xP": print-last-kbd-macro

# M-m is back-to-indentation which is what I usually use to
go to the
# beginning of a line; everywhere else, I bind it like C-a.
"\em": beginning-of-line
# M-p and M-n should behave like they do in M-x shell in
Emacs.
"\ep": previous-history
"\en": next-history

# C-M-f and C-M-v dump functions and variables.
"\e\C-f": dump-functions
"\e\C-v": dump-variables
```

Conclusion

GNU readline's a real gem, and underused. I hope more people – particularly Emacs users, as the keys are so familiar – learn a little bit more about readline and its capabilities. Memorizing a handful of key bindings will supercharge your command line productivity.

Further Reading

Have you read my Reading Guide (/reading-guide) yet? It's a curated guide to most of my articles and I guarantee you'll learn something whether you're a beginner or an expert. And why not check out my book? (/book)

I write infrequently, so go on — sign up and receive an e-mail when I write new articles

Your E-mail address

Your First name

Would you like to try a sample of *Mastering Emacs* for free?

Subscribe

Very useful, thanks! I liked the keyboard macros and dabbrev-expand in particular.

—*Mathias* · reply

Thanks for the post, very useful stuff. Another useful readline variable is completion-ignore-case. Enabling this option makes working with mixed case filenames way easier.

—*Michael* · reply

Is there a reason for the macro to wrap with ezf to be "\C-e | ezf -f 1)\C-a(\C-a$\C-b\C-a", rather than "\C-e | ezf -f 1)\C-a $(\C-a"? Going to the start of the line, adding the"$(" and then returning to the start of the line seems more straight forward than inserting a "(", then inserting a "$" before it, and the \C-b seems redundant.

I think an extra space is also helpful before the "$(".

> An alternative to setting keys to call dump-functions and dump-variables, bash provides the "bind" command which takes options "-p" and "-v". It is simple to search this, e.g. with grep,
>
> I think it is worth mentioning that M-. takes note of numeric prefixes so you can get the first argument of the previous command by M-1 M-.

—*Icarus* · reply

> Very useful post as always. I found macros, dynamic abbrev, yank/cycle kill ring and env. variable expansion to be useful. I've been using Bash for more than 8 years and didn't know about them :)

—*Sundaram (https://legends2k.github.io/)* · reply

Name

Email

Website

`C-x C-f` is what command?

Comment Content

Comment Cancel