Reading and Writing Buffers In Practice

Why learn about buffers?

You may think buffers are just for editing text files, but you can wield Emacs Lisp to do so much more:

- Automate the editing (or creation) of files
- Create a custom display of information for a major mode
- Communicate with external programs
- Provide interactive interfaces like REPLs and more

In Emacs, buffers are the primary user interface, so they're really important!

What will we learn?

In this episode we're going to focus on the basic APIs for Emacs buffers and then show how to read and edit buffer contents.

We're going to use a concrete example as the basis for everything we learn!

Our project: dotcrafter.el

As we talked about in previous videos in this series, we are working on a dotfiles management package for Emacs.

I created a repo for it! https://github.com/daviwil/dotcrafter.el. Check it out and give it a star!

In this video, we'll update our dotfiles management package code to add the following capabilities:

- Automatically detect tangled configuration output files from our Org Mode files
- Update the dotfiles repository's `.gitignore` file to add the tangled output files so that they don't get checked in

Why?

The goal of this package is to manage configuration files that are generated from Org Mode code blocks.

The pattern is to generate these files into the dotfiles folder and then symbolically link them into your home directory.

We might not want to check in those files since we will have to keep them in sync! Instead, we add them to `.gitignore` in the dotfiles repo.

The code we will write today will automatically update the `.gitignore` file with the generated configuration file paths.

What is a buffer?

In Emacs Lisp, a buffer is an object that contains text to be displayed, edited, or manipulated. The content of buffer may not come from a file, it could be generated from within Emacs!

The text isn't just plain letters, it can also contain additional properties that control font, color, size, and other factors!

In Emacs, a buffer is usually displayed by a window, but you can also work with buffers without displaying them.

As we learned in the video about variables, a buffer can have it's own set of variables!

Getting the "current" buffer

Emacs' buffer manipulation functions work on the current buffer.

In Emacs Lisp, you can get the current buffer with the aptly-named `current-buffer` function:

```
(current-buffer)
```

The current buffer in this case is not necessarily the one visible in the selected window!

Getting a buffer by name

If you want to retrieve a buffer by its name, use `get-buffer` :

```
(get-buffer "*scratch*")
```

You can also create the buffer if it doesn't already exist with `get-buffer-create` :

```
(get-buffer-create "Hello System Crafters!")
```

You can now see this in the buffer list!

Changing the current buffer

You can change the current buffer with `set-buffer` . This allows you to set the current buffer that the buffer manipulation functions will operate on:

```
(progn
  (set-buffer (get-buffer "*scratch*"))
  (current-buffer))

;; You can also pass the name of the buffer directly
(progn
  (set-buffer "*scratch*")
  (current-buffer))
```

Changing the current buffer safely!

However, this sets the current buffer until the current command in the command loop is

```
(progn
  (save-current-buffer
    (set-buffer "*scratch*")
    (message "Current buffer: %s" (current-buffer)))
  (current-buffer))
```

For an even shorter solution, you can use the `with-current-buffer` macro:

```
(progn
  (with-current-buffer "*scratch*"
    (message "Current buffer: %s" (current-buffer)))
  (current-buffer))
```

If you use `set-buffer` in your code, you almost always want to use `save-current-buffer` or `with-current-buffer` so that you don't cause weird things to happen in Emacs! These functions also handle errors correctly and ensure that the previous buffer is set back to what it was before you changed it.

Working with file buffers

Most often in Emacs, you'll be working with buffers that contain text loaded from a file. If Emacs created the buffer (using `find-file` or a similar function), you can use the `buffer-file-name` function to get full file path for the file that the buffer represents:

```
(buffer-file-name)
```

You can also find a buffer that represents a particular file (or file path) using the `get-file-buffer` function:

```
(get-file-buffer "Emacs-Lisp-04.org")
(get-file-buffer "~/Notes/Streams/Emacs-Lisp-05.org")
```

```
(get-file-buffer "~/Notes2/Streams/Emacs-Lisp-05.org")
```

NOTE: This function will convert the path you provide into its absolute file path before searching for it. Path expansion takes the `default-directory` variable into account, so make sure to provide as much path context as you can to make sure the right file is found!

Example: Getting the buffers for our configuration Org files

```
(setq dotcrafter-org-files '("Emacs.org" "Desktop.org" "Systems.org"))

(dolist (org-file dotcrafter-org-files)
  (with-current-buffer (get-file-buffer (expand-file-name org-file
                                                          dotcrafter-do
    (message "File: %s" (buffer-file-name))))
```

Loading a file into a buffer

Emacs provides a function for "visiting" a file without displaying it, mainly for opening a file into a buffer. This function is called `find-file-noselect`:

```
(find-file-noselect "Emacs-Lisp-01.org")
```

A couple of interesting details:

- Running this more than once for the same file will return the same buffer
- If the buffer for that file is modified and not saved, the user **might** be prompted before opening a new buffer for the file
- To prevent the user from being prompted, send `t` as the second argument:

```
(find-file-noselect "Emacs-Lisp-01.org" t)
```

Example: Getting or creating the Org file buffer

We can use `find-file-noselect` to create a buffer for the file if it doesn't already exist:

```
(dolist (org-file dotcrafter-org-files)
  (let ((file-path (expand-file-name org-file
                                     dotcrafter-dotfiles-folder)))
    (with-current-buffer (or (get-file-buffer file-path)
                             (find-file-noselect file-path))
      (message "File: %s" (buffer-file-name)))))
```

What's the point?

The point of talking about all of this is so that we can talk about the point!

The "point" is the location of the cursor in the buffer. It is the location from which all hand editing commands operate.

The point is represented by an integer (whole number) which starts at 1 and increases for every character in the buffer.

```
(point)
```

NOTE: The same buffer can be displayed in multiple windows and the point can be different in those two windows!

You can also check the minimum and maximum point locations of the buffer using `point-min` and `point-max`:

```
(point-min)
(point-max)
```

"Narrowing" may affect these positions, but we'll talk about that another time.

Moving the point

You can use the following motion commands to move the point:

- `goto-char` - Move the point to a specific position (integer)
- `forward-char` - Move the point forward by a number of positions (1 by default)
- `backward-char` - Move the point backward by a number of positions (1 by default)

- `beginning-of-buffer` - Go to the beginning of the buffer
- `end-of-buffer` - Go to the end of the buffer

You can also move based on larger textual units in the buffer:

- `forward-word` - Move forward by one “word”
- `backward-word` - Move backward by one “word”

```
(goto-char 1)
(goto-char (point-min))
(goto-char (point-max))
(beginning-of-buffer)
(end-of-buffer)

(forward-char)
(forward-char 5)

(backward-char)
(backward-char 10)

(forward-word)
(backward-word)
```

There are many more `forward` and `backward` functions, just check the function list (`describe-function` , `C-h f`) to find them!

The cool thing about these functions is that they’re the same ones you use with Emacs default movement keybindings. You can automate buffer editing with the same functions you use for typing!

Preserving the point

Similarly to setting the current buffer, you might want to preserve the current point location in a buffer before you move it for another purpose. You can use the `save-excursion` special form for this purpose:

```
(save-excursion
  (goto-char (point-max))
  (point))
```

This is useful when you need to do an operation in the current buffer which might be displayed in the user’s current window!

Examining buffer text

Now that we understand the point, we can talk about how to look at text in the buffer. The simplest thing you can do is read the character at a location with `char-after`:

```
(char-after)
(char-after (point))
(char-after (point-min))
```

You can also get a substring of text in the buffer between two points using `buffer-substring` and `buffer-substring-no-properties`:

```
(buffer-substring 9328 9349)
(buffer-substring-no-properties 9328 9349)
```

The Thing

The `thing-at-point` function is very useful for grabbing the text at the point if it matches a particular type of "thing":

- `word`, `sentence`, and `line`
- `sexp`, `list`, and `defun` - Lisp expressions
- `url`, `email`
- `filename`

```
(thing-at-point 'word)
(thing-at-point 'sentence)
(thing-at-point 'sentence t)
(thing-at-point 'sexp)
```

Try it on this: ~/Projects/Code/emacs-from-scratch/Emacs.org

Emacs Lisp Manual: Examining Buffer Contents

Searching for text

Sometimes it can be useful to search for text inside of the buffer and move the point to where the match was found. You can use the `search-forward` and `search-backward` functions for this:

```
(search-forward "ways")
(search-backward "I just searched myself")
(search-backward "inside" nil t 1)
(search-backward "inside" nil t 3)
```

Keep in mind that `search-forward` will put the point **after** the match and `search-backward` will put the point **before** the match!

The other parameters can be useful too:

- `bound` (param 2) - A pair (cons) of positions restricting the search within those two positions
- `noerror` - If `t`, don’t signal an error when no match is found
- `count` - Find the “nth” result where `count` is `n`

There are other ways to search inside of buffers, including the use of regular expressions to extract text from matches. We’ll cover this in another episode!

Example: Finding Org code block output paths

Now we can finally do something useful in our code! Let’s write some Emacs Lisp to search for `:tangle` properties on Org source blocks so that we can extract the file path:

```
(defun dotcrafter--scan-for-output-files (org-file)
  (let ((output-files '())
        (current-match t))
    (with-current-buffer (or (get-file-buffer org-file)
                             (find-file-noselect org-file))
      (save-excursion
        (goto-char (point-min))  ;; Or (beginning-of-buffer)
        (while current-match
          (setq current-match (search-forward ":tangle " nil t))
          (when current-match
            (let ((output-file (thing-at-point 'filename t)))
              ;; If a file path was found, add it to the list
              (unless (or (not output-file)
                          (string-equal output-file "no"))
```

```
                  (setq output-files (cons output-file
                                           output-files))))))))
    output-files))

(let ((output-files '()))
  (dolist (org-file dotcrafter-org-files)
    (setq output-files
          (append output-files
                  (dotcrafter--scan-for-output-files
                   (expand-file-name org-file
                                     dotcrafter-dotfiles-folder)))))

  output-files)
```

Inserting text

You can insert text into the buffer at the current point using the `insert` and `insert-char` functions. `insert` will insert the arbitrary list of strings or characters at point and `insert-char` will insert the specified character with an optional repeat count:

```
(insert "  0_o")
(insert "\n" "This is" ?\s ?\n "Sparta!")

(insert-char ?\- 20)
```

Emacs Lisp Manual: Inserting Text Emacs Lisp Manual: Basic Char Syntax

Example: Updating the .gitignore file

```
(defvar dotcrafter-gitignore-marker "\n# -- Generated by dotcrafter.el!
  "The marker string to be placed in the .gitignore file of the
dotfiles repo to indicate where the auto-generated list of ignored
files begins.")

(defun dotcrafter--update-gitignore ()
  (let ((output-files '()))
    (dolist (org-file dotcrafter-org-files)
      (setq output-files
            (append output-files
                    (dotcrafter--scan-for-output-files
                     (expand-file-name org-file
                                       dotcrafter-dotfiles-folder)))))
    (let ((gitignore-file (expand-file-name ".gitignore"
                                            dotcrafter-dotfiles-folder)
      (with-current-buffer (or (get-file-buffer gitignore-file)
                               (find-file-noselect gitignore-file))
        (save-excursion
          (beginning-of-buffer)
          (or (progn
                (search-forward dotcrafter-gitignore-marker nil t))
              (progn
                (end-of-buffer)
                (insert "\n" dotcrafter-gitignore-marker)))

          (dolist (output-file output-files)
            (insert output-file "\n")))))))
```

Notice that running this multiple times appends the list again and again! We need to delete the old list before adding the new one.

Deleting text

You can delete a region of text in a buffer using the `delete-region` function. It takes two parameters, the `start` point and the `end` point.

```
(with-current-buffer ".gitignore"
  (delete-region (point) (point-max)))
```

Saving a buffer

To save the contents of a buffer back to the file it is associated with, you can use the `save-buffer` function:

```
(save-buffer)
```

Example: Cleaning up and saving the .gitignore file

Let's finish the job of automatically managing the `.gitignore` file by cleaning up its contents and saving it:

```
(defun dotcrafter--update-gitignore ()
  (let ((output-files '()))
    (dolist (org-file dotcrafter-org-files)
      (setq output-files
            (append output-files
                    (dotcrafter--scan-for-output-files
                     (expand-file-name org-file
                                       dotcrafter-dotfiles-folder)))))

    (let ((gitignore-file (expand-file-name ".gitignore"
                                            dotcrafter-dotfiles-folder)))
      (with-current-buffer (or (get-file-buffer gitignore-file)
                               (find-file-noselect gitignore-file))
        (save-excursion
          (beginning-of-buffer)
          (or (progn
                (search-forward dotcrafter-gitignore-marker nil t))
              (progn
                (end-of-buffer)
                (insert "\n" dotcrafter-gitignore-marker)))

          (delete-region (point) (point-max))
          (dolist (output-file output-files)
            (insert output-file "\n"))

          (save-buffer))))))
```

What's next?

In the next episode, we'll cover how to manage files and directories in Emacs Lisp. We'll also extend `dotcrafter` to create symbolic links to their target locations in the home directory!

Final Code

You can check out the final code here at the `dotcrafter.el` repository!

I've also included the code below for posterity:

```
(defvar dotcrafter-gitignore-marker "\n# -- Generated by dotcrafter.el!
  "The marker string to be placed in the .gitignore file of the
dotfiles repo to indicate where the auto-generated list of ignored
files begins.")

(defun dotcrafter--scan-for-output-files (org-file)
  (let ((output-files '())
        (current-match t))
    ;; Get a buffer for the file, either one that is
    ;; already open or open a new one
    (with-current-buffer (or (get-file-buffer org-file)
                             (find-file-noselect org-file))
      ;; Save the current buffer position
      (save-excursion
        ;; Go back to the beginning of the buffer
        (goto-char (point-min))

        ;; Loop until no more matches are found
        (while current-match
          ;; Search for blocks with a :tangle property
          (setq current-match (search-forward ":tangle " nil t))
          (when current-match
            (let ((output-file (thing-at-point 'filename t)))
              ;; If a file path was found, add it to the list
              (unless (or (not output-file)
                          (string-equal output-file "no"))
                (setq output-files (cons output-file
                                         output-files)))))))))
    output-files))

(defun dotcrafter--update-gitignore ()
  (let ((output-files '()))
    ;; Gather the list of output files from all Org files
    (dolist (org-file dotcrafter-org-files)
      (setq output-files
            (append output-files
                    (dotcrafter--scan-for-output-files
                     (expand-file-name org-file dotcrafter-dotfiles-fol

    ;; Now that we have the output files, update the .gitignore file
    (let ((gitignore-file (expand-file-name ".gitignore"
                                            dotcrafter-dotfiles-folder)
      ;; Find the .gitignore buffer and prepare for editing
      (with-current-buffer (or (get-file-buffer gitignore-file)
                               (find-file-noselect gitignore-file))
        (save-excursion
          ;; Find or insert the dotcrafter-gitignore-marker
          (beginning-of-buffer)
          (or (progn
                (search-forward dotcrafter-gitignore-marker nil t))
              (progn
                (end-of-buffer)
                (insert "\n" dotcrafter-gitignore-marker)))

          ;; Delete the rest of the buffer after the marker
          (delete-region (point) (point-max))

          ;; Insert a line for each output file
          (dolist (output-file output-files)
            (insert output-file "\n"))

          ;; Make sure the buffer is saved
          (save-buffer))))))
```