Table of Contents

Using the computer to take notes makes it easy to search and share the information. Emacs has a number of powerful text-editing tools that let you quickly capture notes, work with outlines, keep daily notes, and post to a blog. Start by reading the section on Remember, then jump to either Outline Notes

with Org or Daily Notes with Planner depending on the kinds of notes you typically take. The last section on blogging from Emacs is useful for integrating with external systems.

Remember

Ideas come from everywhere. While reading this blog, you might come across interesting snippets that you'd like to save. While writing code, you might be hit by an idea for something you want to do with the program. While on a phone call, you might need to write down what you need to prepare for a meeting the next day.

How do you take notes now? Do you jot your notes on a scrap of paper or into a leather notebook? Do you copy and paste what you're looking at into a plain text file or document? Do you scribble things into a personal digital assistant?

I've tried different note-taking strategies: colorful mindmaps, outlined text files, even voice recordings. I felt frustrated every time I had to write down whose e-mail or which webpage prompted the note. Shouldn't the computer do that automatically? I was even more frustrated when I'd come across a note and not remember why I wrote it.

Remember changed all that for me. The key ideas behind Remember are that you should be able to write notes with minimal distraction, and that the context should be automatically picked up so that you don't have to write it down. If you're writing a note based on a mail message, Remember will pick up the details of the message and create a hyperlink so that you can view the original message when reviewing your notes. If you're working on a file, Remember will link to it so that you can jump back to it. If you're browsing a web page in Emacs, Remember will remember the title and URL, so that you can go back to the page when you want to clarify something in your notes.
After you save the note, you'll be back in the same environment you were: no need to switch applications and no need to remember different keyboard shortcuts.

You might think that Remember is only worth using if you do *everything* in Emacs. For me, it worked the other way around. I started by using Remember to take notes in Planner, a personal information manager available for Emacs. As I got accustomed to the way Remember and Planner just automatically hyperlinked to whatever I was looking at, I thought: Why doesn't my mail client do this? Why doesn't my web browser do this? Why doesn't my chat client do this? So I ended up reading through the manuals, figuring out how to do all these things in Emacs—and I loved it, eventually doing most of my work and play within an Emacs frame. Although I use other applications now, like Lotus Notes for work mail and Mozilla Firefox for browsing, I still switch back to Emacs for my notes.

In this section, you'll learn how to set up Remember and take quick notes in Emacs. We'll start by configuring Remember to save your notes to a file, and how to configure Remember to save to different places depending on the content. You'll also learn how to search your notes file for entries. You can integrate Remember into other note-taking systems in Emacs. The sections that cover those systems will also show you how to configure Remember to save your notes there.

Project XXX: Set up Remember

Remember is a separate package, which you can download from *https://gna.org/projects/remember-el* . As of this writing, the latest stable release is Remember 1.9. Download the latest version and unpack it into your *~/elisp* directory. You should end up with a new directory, *~/elisp/remember-1.9* .

To configure Remember to save to plain text files, add this code to your *~/.emacs* and evaluate it:

```
(add-to-list 'load-path "~/elisp/remember-1.9") ❶
(require 'remember-autoloads)
(setq remember-data-file "~/notes.txt")  ❷
(global-set-key (kbd "C-c r") 'remember) ❸

(defun wicked/remember-review-file ()
 "Open `remember-data-file'."
 (interactive)
 (find-file-other-window remember-data-file))
(global-set-key (kbd "C-c R") 'wicked/remember-review-file) ❹
```

First, you need to❶add the directory to your load path. You will also need to ❷choose a file for your notes. **C-c r** (`remember`)❸is a handy shortcut key for remember. You can also bind it to other shortcut keys such as **F9 r**. Lastly, ❹bind `wicked/remember-review-file` to a shortcut that makes it easy to check your remembered notes.

After you've configured Remember, try it out by typing **C-c r** (`remember`). Your Emacs frame will be split in two, and one of the windows will be a *Remember* buffer. Type your note. The first line will be the headline, and the rest of the buffer will be the body of the note. If you call **C-c r** (`remember`) from a file, the filename will automatically be included at the end of the buffer. Type **C-c C-c** (`remember-buffer`) to save the note.

If you open *~/notes.txt* to review your note, you'll find something like this:

```
** Sat Jan 12 14:43:02 2008 Headline goes here
```

```
Note body goes here

The filename goes here
```

To remember a region of text, use **C-u C-c r** (`remember`). The selected text will be included in the buffer, so you don't need to copy and paste it yourself.

Make a habit of typing **C-c C-r** or **C-u C-c C-r** (`remember`) when you need to remember something. Type **C-c C-c** (`remember-buffer`) to get back to work, knowing that your notes have been safely saved in your *~/notes.txt* file.

Project XXX: Review Your Notes

Use **C-c R** (`wicked/remember-review-file`) to check your notes, or open *~/notes.txt* yourself.

To search your notes, use **C-c R** (`wicked/remember-review-file`) to open the file, then use **C-s** (`isearch-forward`) to search for words interactively, or use **M-x occur** to find all lines containing a word.

You may notice that the default format that Remember uses is an outline format that is compatible with Org and Allout, both of which have powerful outline-editing tools. I prefer Org's outline-editing commands, and you'll learn about them in the "Outline Notes with Org" section.

You can treat *~/notes.txt* as your inbox, and keep your organized notes in another file or groups of files. Cut and paste the text between the files to clear your inbox, and use **M-x grep** to search multiple files. Alternatively, you can keep all of your notes in one large text file, and use **C-s** (`isearch-forward`) and **M-x occur** to search for information.

Now you know the basics of remembering information, saving it into a file, and reviewing the file. By default, Remember annotates your notes with a filename if you were looking at a file when you called **C-c r** (`remember`). As you learn more about Emacs, you may want to configure Remember to add more intelligent annotations and other text to the Remember buffer. The more work Remember does for you, the less work you have to do!

Project XXX: Set up Annotation Functions

The easiest way to get Remember to automatically understand mail messages, Web pages, info files, BBDB contact records, and other sources of information in Emacs is to use either Org or Planner. To learn how to integrate Remember with either Org or Planner, read the section on "Outline Your Notes with Org" and "Write Your Journal with Planner".

You can also define your own annotation functions. When you call **C-c r** (`remember`) from a buffer, Remember goes through each of the functions in `remember-annotation-functions`, and it uses the first non-nil value returned.

For example, you may work with many temporary buffers that don't have filenames. To create an annotation function that adds buffer names, add the following code to your *~/.emacs* after the basic Remember configuration code:

```
(eval-after-load 'remember
  '(progn
    (add-to-list 'remember-annotation-functions 'buffer-name t)))
```

This adds `buffer-name` to the end of the annotation list, making it a last resort.

What if you want line numbers included with the filename or buffer name? You could replace the previous code with this:

```
(defun wicked/remember-line-numbers-and-file-names ()
 "Return FILENAME line NUMBER."
 (save-restriction
  (widen)
  (format " %s line %d"
   (or (buffer-file-name) (buffer-name))
   (line-number-at-pos))))
(eval-after-load 'remember
  '(progn
     (add-to-list 'remember-annotation-functions
                  'wicked/remember-line-numbers-and-file-names)))
```

With that code, **C-c r** (`remember`) will automatically pick up the line number from your file.

By default, Remember saves your notes to a plain-text file, so you'll have to open the files manually. The command **M-x ffap** or `find-find-file-at-point` may be convenient. If you want hyperlinks that you can visit easily, consider saving your notes in an Org or Planner file instead.

Now you've got context. What else can you do with the Remember buffer?

Project XXX: Add Other Text to the Remember Buffer

Remember has plenty of hooks that let you modify the behavior. For example, you might want to insert a random tagline or fortune-cookie saying whenever you create a note. This is a fun way to encourage yourself to write more, because there will be a little surprise every time you open a Remember buffer.

Here is a totally small-scale way to use random lines from a text file. Let's say that you have a text file made up of movie quotes, taglines, knock-knock jokes, or short fortune-cookie sayings. When I wrote this code, I used Japanese/English sentence pairs about cats, because I was studying Japanese. You can use whatever tickles your fancy, as long as this text file (*~/taglines.txt*) has one line per saying.

ch07-random-tagline.el:

```
(defun wicked/random-tagline (&optional file)
  "Return a random tagline."
  (with-current-buffer (find-file-noselect (or file "~/taglines.txt"))
    (goto-char (random (point-max)))
    (let ((string
           (buffer-substring (line-beginning-position)
                             (line-end-position))))
      string)))

(eval-after-load 'remember
  '(progn
     (defadvice remember (after wicked activate)
       "Add random tagline."
       (save-excursion
         (goto-char (point-max))
         (insert "\n\n" (wicked/random-tagline) "\n\n")))))
```

If you want multi-line sayings, look into the Emacs fortune cookie package, and replace `wicked/random-tagline` with a function that returns a random string.

This code modifies the behavior of **C-c r** (`remember`) by inserting a random tagline after the buffer has been prepared. You can use the same idea to insert a timestamp noting you started, a template figuring or modify the text in other ways.

Project XXX: Save to Different Places

You can change how Remember saves its notes. For example, if you want all of the notes that contained `:EMACS:` or `:WORK:` to go into separate files, you can add this code to your *~/.emacs*:

ch07-remember-keywords.el:

```
(defvar wicked/remember-keywords
  '((":EMACS:" . "~/emacs.txt")
    (":WORK:" . "~/work.txt"))
```

```
  "*List of (REGEXP . FILENAME).
If an entry matches REGEXP, it will be storied in FILENAME.
The first regular expression that matches is used.")
(eval-after-load 'remember
  '(progn
     (defadvice remember-region (around wicked activate)
       "Save notes matching `wicked/remember-keywords' elsewhere."
       (let* ((b (or beg (min (point) (or (mark) (point-min)))))
              (e (or end (max (point) (or (mark) (point-max)))))
              (string (buffer-substring-no-properties b e))
              (done nil)
              (keywords wicked/remember-keywords))
         (while keywords
           (when (string-match (caar keywords) string)
             (let ((remember-data-file (cdar keywords)))
               ad-do-it)
             (setq keywords nil done t))
           (setq keywords (cdr keywords)))
         (unless done
           ad-do-it)))))
```

You can configure Remember to use different handler functions. This chapter covers several note-taking systems for Emacs, and you may want to use Remember to save to more than one note-taking system. For example, you can set up **C-c r p** to start a Remember buffer that saves to Planner, and **C-c r o** to start a Remember buffer that saves to Org. Here's the code for your *~/.emacs*:

```
(defun wicked/remember-to-org ()
  "Remember to Org."
  (let ((remember-annotation-functions    ❶
         (cons 'org-remember-annotation
               remember-annotation-functions)))
    (remember)
    (set (make-variable-buffer-local
          'remember-handler-functions)
         '(org-remember-handler))))      ❷

(defun wicked/remember-to-planner ()
  "Remember to Planner."
  (let ((remember-annotation-functions    ❸
         (append planner-annotation-functions
```

```
            remember-annotation-functions)))
    (remember)
    (set (make-variable-buffer-local
        'remember-handler-functions)
       '(remember-planner-append))))      ❹

(global-unset-key (kbd "C-c r"))  ❺
(global-set-key (kbd "C-c r o") 'wicked/remember-to-org)
(global-set-key (kbd "C-c r p") 'wicked/remember-to-planner)
```

This code ensures that when you call `wicked/remember-to-org`, Emacs will ❶ insert Org-compatible links, and❷save notes started with wicked/remember-to-org to an Org file. Likewise, `wicked/remember-to-planner` ❸will use Planner-compatible links and ❹save notes to Planner pages. We then bind the functions to handy keyboard shortcuts. Before you can bind a keyboard shortcut that starts with **C-c r** (currently bound to `remember`), you need to ❺ unset the existing keyboard shortcut. Then you can bind **C-c r o** to `wicked/remember-to-org` and **C-c r p** to `wicked/remember-to-planner`.To learn more about the different note-taking systems, read on.

Outline Notes with Org

Large documents are almost impossible to write without outlines. There's just too much to fit in your head. Outlines help you work with a structure, so that you can see the big picture and how sections fit together. Outlines are also surprisingly useful when brainstorming. You can work with varying levels of detail, starting with a high-level overview and successively refining it, or starting with the details and then letting the structure emerge as you organize those details into groups.

Emacs has one of the most powerful outline editors I've come across. Although word processors like Microsoft Word and OpenOffice.org Writer support outlines too, Emacs has a gazillion keyboard shortcuts, and once you get the hang of them, you'll want them in other applications as well. In addition, working in Emacs tends to force you to focus on the content instead of spending time fiddling with the formatting. Whether you're writing personal notes or working on a document that's due tomorrow, this is a good thing.

In this section, you'll learn how to outline a document using Org. You'll be able to create headings, sub-headings, and text notes. You'll also learn how to manage outline items by promoting, demoting, and rearranging them.

Project XXX: Set up Org

Org is primarily a personal information manager that keeps track of your tasks and schedule, and you'll learn more about those features in Chapters 8 and 9. However, Org also has powerful outline-management tools, which is why I recommend it to people who want to write and organize outlined notes.

The structure of an Org file is simple: a plain text file with headlines, text, and some additional information such as tags and timestamps. A headline is any line that stars with a series of asterisks. The more asterisks there are, the deeper the headline is. A second-level headline is the child of the first-level headline before it, and so on. For example:

```
* This is a first-level headline
Some text under that headline
** This is a second-level headline
Some more text here
*** This is a third-level headline
*** This is another third-level headline
** This is a second-level headline
```

Because Org uses plain text, it's easy to back up or process using scripts. Org's sophistication comes from keyboard shortcuts that allow you to quickly manipulate headlines, hide and show outline subtrees, and search for information.

GNU Emacs 22 includes Org, but you can download an updated version from the Org homepage *http://orgmode.org/*. As of this writing, the latest version of Org is 5.19, and you can download it from *http://orgmode.org/org-5.19a.tar.gz* or *http://orgmode.org/org-5.19a.zip*. To automatically enable Org mode for all files with the .org extension, add the following to your *~/.emacs*:

```
(require 'org)
(add-to-list 'auto-mode-alist '("\\.org$" . org-mode))
```

To change to `org-mode` manually, use **M-x org-mode** while viewing a file. To enable Org on a per-file basis, add

```
*- mode: org -*-
```

to the first line of the files that you would like to associate with Org.

Project XXX: Work with Outlines

You can keep your notes in more than one Org file, but let's start by working with one. Create *~/notes.org*, which will automatically be associated with Org Mode. Type in the general structure of your document. You can type in something like this:

```
* Introduction
* ...
```

You can also use **M-RET** (`org-meta-return`) to create a new headline at the same level as the one above it, or a first-level headline if the document doesn't have headlines yet.

Create different sections for your work. For example, a thesis might be divided into introduction, review of related literature, description of study, methodology, results and analysis, and conclusions and recommendations. You can type in the starting asterisks yourself, or you can use **M-RET** (`org-meta-return`) to create headings one after the other.

Now that you have the basic structure, start refining it. Go to the first section and use a command like **C-o** (`open-line`) to create blank space. Add another headline, but this time, make it a second-level headline under the first. You can do that by either typing in two asterisks or by using **M-RET** (`org-meta-return`, which calls `org-insert-heading`) and then **M-right** (`org-metaright`, which calls `org-do-demote`). Then use **M-RET** (`org-meta-return`; `org-insert-heading`) for the other items, or type them in yourself. Repeat this for other sections, and go into as much detail as you want.

What if you want to reorganize your outline? For example, what if you realized you'd accidentally put your conclusions before the introduction? You could either cut and paste it using Emacs shortcuts, or you can use Org's outline management functions. The following keyboard shortcuts are handy:

Action	Command	Shortcut	Alternative
Move a subtree up	org-metaup (org-move-subtree-up)	M-up	C-c C-x u
Move a subtree down	org-metadown (org-move-subtree-down)	M-down	C-c C-x d
Demote a subtree	org-shiftmetaright (org-demote-subtree)	S-M-right	C-c C-x r
Promote a subtree	org-shiftmetaleft (org-promote-subtree)	S-M-left	C-c C-x l
Demote a headline	org-metaright (org-do-demote)	M-right	C-c C-x right
Promote a headline	org-metaleft (org-do-promote)	M-left	C-c C-x left
Collapse or expand a subtree	org-cycle while on a headline	TAB	
Collapse or expand everything	org-shifttab	S-TAB	C-u TAB

Table 1: Keyboard shortcuts for outline manipulation

Use these commands to help you reorganize your outline as you create a skeleton for your document. These commands make it easy to change your mind about the content or order of sections. You might find it easier to sketch a rough outline first, then gradually fill in more and more detail. On the other hand, you might find it easier to pick one section and keep drilling it down until you have headlines some seven levels deep. When you reach that point, all you need to do is add punctuation and words like "and" or "but" between every other outline item, and you're done!

Well, no, not likely. You'll probably get stuck somewhere, so here are some tips for keeping yourself going when you're working on a large document in Org: brainstorming and getting a sense of your progress.

Project XXX: Brainstorm

Brainstorming is a great way to break your writer's block or to generate lots of possibilities. The key idea is to come up with as many ideas as you can, and write them all down before you start evaluating them. I usually switch to paper for mindmapping and brainstorming because paper helps me think in a more colorful and non-linear way. However, it can be hard to manage large mindmaps on paper, because you can't reorganize nodes easily. Despite its text-heavy interface, Org is one of the best

mindmapping tools I've come across. Because it's built into Emacs, everything can be done through keyboard shortcuts.

When you're brainstorming, you might like working from two different directions. Sometimes it's easier to start with an outline and to add more and more detail. Other times, you may want to jot quick ideas and then organize them into groups that make sense. Org provides support for both ways of working.

Brainstorming bottom-up is similar to David Allen's Getting Things Done method in that separating collection from organization is a good idea. That is, get the ideas out of your head first before you spend time trying to figure out the best way to organize them. To get things out of your head quickly, collect your ideas by using the **M-RET** (`org-meta-return`) to create a new heading, typing in your idea, and using **M-RET** (`org-meta-return`) to create the next heading. Do this as many times as needed.

One way to encourage yourself to brainstorm lots of ideas is to give yourself a quota. Charles Cave described this technique in an article on org-mode, and it's a great way to use structure to prompt creativity. Simply start by adding a number of empty headings (`say, 20`), then work towards filling that quota. For example, you might start with ten blanks for ideas, then gradually fill them in like this:

```
* Things that make me happy
** Curling up with a good book
** Watching a brilliant sunset
** Giving or getting a big warm hug
** Writing a cool piece of Emacs code
**
**
**
**
**
**
```

When all of your ideas are in Org, start organizing them. This is where you move ideas around using **M-S-up** (`org-shiftmetaup`) and **M-S-down** (`org-shift-metadown`), which call org-move-subtree-up and org-move-subtree-down. This is also a good time to use headings to group things together.

Project XXX: Get a Sense of Progress

You've brainstormed. You've started writing your notes. And if you're working on a large document, you might lose steam at some point along the way. For example, while working on this book, I often find myself intimidated by just how much there is to write about Emacs. It helps to have a sense of progress such as the number of words written or the number of sections completed. To see a word count that updates every second, add this code to your *~/.emacs*:

ch07-count-words.el

```
(defun wicked/count-words-buffer () ❶
  "Return the number of words in the current buffer."
  (save-excursion
    (goto-char (point-min))
    (let ((count 0))
      (while (not (eobp))
        (forward-word 1)
        (setq count (1+ count)))
      count)))

(defvar wicked/count-words-buffer nil "*String to display.") ❷
(defvar wicked/count-words-goal nil "*Target number of words.")

(defun wicked/update-word-count ()
  "Update the displayed word count."
  (interactive)
  (setq wicked/count-words-buffer  ❸
    (if wicked/count-words-goal
       (format "%d word(s) left" (- wicked/count-words-goal
                                    wicked/count-words-buffer))
     (format "%d word(s)" wicked/count-words-buffer))
  (force-mode-line-update))

;; Add the string to the mode line
(unless (memq 'wicked/count-words-buffer global-mode-string) ❹
  (add-to-list 'global-mode-string 'wicked/count-words-buffer t))

;; Update the modeline when idle
(unless wicked/count-words-buffer  ❻
  (run-with-idle-timer 1 t 'wicked/update-word-count))
```

```
(defun wicked/count-words-update-quota (&optional words)
  (interactive "nWords: ")
  (setq wicked/count-words-goal  ❺
        (+ (wicked/count-words-buffer)
           (or words 2000))))
```

Counting words is as simple as❶going to the beginning of the buffer and moving the cursor forward one word at a time, keeping track of the number of words seen. This avoids the need to call a separate program such as *wc* (word count). The❷`wicked/count-words-buffer` and `wicked/count-words-goal` store information so that it can be displayed in the modeline. The wicked/update-word-count❸sets the `wicked/count-words-buffer` string to either the difference between the current number of words and the target number of words, or the current number of words. ❹ A placeholder is added to the modeline, and a timer ❺ updates the placeholder periodically.

To set the goal number of words, use `wicked/count-words-update-quota`, which prompts you for the number of words you want to write today. It then❹calculates the target number of words based on the current total. With that, you'll have a good idea of your progress in terms of word count.

The best way I've found to track my progress in terms of sections is to use a checklist. For example, the collapsed view for this section looks like this:

```
** Outline Notes with Org [7/9]    ❶
*** [X] Outlining Your Notes...    ❷
*** [X] Understanding Org...
*** [X] Working with Outlines...
*** [X] Brainstorming...
*** [ ] Getting a Sense of Progress... ❸
*** [X] Searching Your Notes...
*** [X] Hyperlinks...
*** [X] Publishing Your Notes...
*** [ ] Integrating Remember with Org
```

The ❶ [7/9] shows that I've completed 7 of 9 parts, ❷[X] indicates finished parts, and ❸ [ ] indicates parts I still need to do. This is based on the checklist feature in Org. However, the standard feature doesn't work with headlines, only plain lists like this:

```
*** Items [1/3]
- [X] Item 1
- [ ] Item 2
- [ ] Item 3
```

Add the following code to your *~/.emacs* in order to make the function work with headlines:

ch07-update-checkbox-count:

```
(defun wicked/org-update-checkbox-count (&optional all)
  "Update the checkbox statistics in the current section.
This will find all statistic cookies like [57%] and [6/12] and update
them with the current numbers.  With optional prefix argument ALL,
do this for the whole buffer."
  (interactive "P")
  (save-excursion
    (let* ((buffer-invisibility-spec (org-inhibit-invisibility))
           (beg (condition-case nil
                    (progn (outline-back-to-heading) (point))
                  (error (point-min))))
           (end (move-marker
                 (make-marker)
                 (progn (or (outline-get-next-sibling) ❶
                            (goto-char (point-max)))
                        (point))))
           (re "\\(\\[[0-9]*%\\]\\)\\|\\(\\[[0-9]*/[0-9]*\\]\\)")
           (re-box
            "^[ \t]*\\(*+\\|[-+*]\\|[0-9]+[.)]\\) +\\(\\[[- X]\\]\\)")
           b1 e1 f1 c-on c-off lim (cstat 0))
      (when all
        (goto-char (point-min))
        (setq beg (point) end (point-max)))
      (goto-char beg)
      (while (re-search-forward re end t)
        (setq cstat (1+ cstat)
              b1 (match-beginning 0)
              e1 (match-end 0)
              f1 (match-beginning 1)
              lim (cond
                   ((org-on-heading-p)
                    (or (outline-get-next-sibling) ❷
                        (goto-char (point-max)))
                    (point))
                   ((org-at-item-p) (org-end-of-item) (point))
                   (t nil))
```

```
            c-on 0 c-off 0)
        (goto-char e1)
        (when lim
          (while (re-search-forward re-box lim t)
            (if (member (match-string 2) '("[ ]" "[-]"))
                (setq c-off (1+ c-off))
              (setq c-on (1+ c-on))))
          (goto-char b1)
          (insert (if f1
                      (format "[%d%%]" (/ (* 100 c-on)
                                          (max 1 (+ c-on c-off))))
                    (format "[%d/%d]" c-on (+ c-on c-off))))
          (and (looking-at "\\[.*?\\]")
               (replace-match ""))))
      (when (interactive-p)
        (message "Checkbox statistics updated %s (%d places)"
                 (if all "in entire file" "in current outline entry")
                 cstat)))))
(defadvice org-update-checkbox-count (around wicked activate) ❸
  "Fix the built-in checkbox count to understand headlines."
  (setq ad-return-value
        (wicked/org-update-checkbox-count (ad-get-arg 1))))
```

This modifies `org-update-checkbox-count` to work with subtrees when it's ❶ determining the start and end of the region to search for the status indicator ([%] or [/]) and when it's ❷searching for [X] markers for completed items.

Now add [ ] or [X] to the lower-level headlines you want to track. Add [/] to the end of the higher-level headline containing those headlines. Type **C-c #** (`org-update-checkbox-count`) to update the count for the current headline, or **C-u C-c C-#** (`org-update-checkbox-count`) to update all checkbox counts in the whole buffer.

If you want to see the percentage of completed items, use `[%]` instead of `[/]`. I find `7/9` to be easier to understand than `71%`, but fractions might work better for other cases.

Okay, you've written a lot. How do you find information again?

Project XXX: Search Your Notes

When you're writing your notes, you might need to refer to something you've written. You may find it helpful to split your screen in two with **C-x 3** (`split-window-horizontally`) or **C-x 2** (`split-`

`window-vertically`). To switch to another window, type **C-x o** (`other-window`) or click on the window. To return to just one window, use **C-x 1** (`delete-other-windows`) to focus on just the current window, or **C-x 0** (`delete-window`) to get rid of the current window.

Now that you've split your screen, how do you quickly search through your notes? **C-s** (`isearch-forward`) and **C-r** (`isearch-backward`) are two of the most useful Emacs keyboard shortcuts you'll ever learn. Use them to not only interactively search your Org file, but also to quickly jump to sections. For example, I often search for headlines by typing * and the first word. Org searches collapsed sections, so you don't need to open everything before searching.

To search using Org's outline structure, use **C-c** / r (`org-sparse-tree, regexp`), which will show only entries matching a regular expression. For more information about regular expressions, read the Emacs info manual entry on Regexps. Here are a few examples:

To find	Search for
All entries containing "cat"	cat
All entries that contain "cat" as a word by itself	\<cat\>
All entries that contain 2006, 2007, or 2008	200[678]

Table 2: Regular expression examples

If you find yourself frequently searching for some sections, you might want to create hyperlinks to them. For example, if you're using one file for all of your project notes instead of splitting it up into one file per project, then you probably want a list of projects at the beginning of the file so that you can jump to each project quickly.

You can also use hyperlinks to keep track of your current working position. For example, if you're working on a long document and you want to keep your place, create a link anchor like `<<TODO>>` at the point where you're editing, and add a link like `[[TODO]]` at the beginning of your file.

You can create a hyperlink to a text search by putting the keywords between two pairs of square brackets, like this:

```
See [[things that make me happy]]
```

You can open the link by moving the point to the link and typing **C-c C-o** (`org-open-at-point`). You can also click on the link to open it. Org will then search for text matching the query. If Org doesn't find an exact match, it tries to match similar text, such as "Things that make me really really happy".

If you find that the link does not go where you want it to go, you can limit the text search. For example, if you want to link to a headline and you know the headline will be unique, you can add an

asterisk at the beginning of the link text in order to limit the search to headlines. For example, given the following text:

```
In this file, I'll write notes on the things that make me happy. ❶
If I ever need a spot of cheering up, I'll know just what to do!

** Things that make me happy ❷
*** ...

Example
Link 1: [[things that make me happy]]

Link 2: [[*things that make me happy]]
```

Link 1 would lead to ❶the first instance of “things that make me happy”, and Link 2 would lead to ❷the headline..

To define a link anchor, put the text in double angled brackets like this:

```
<<things that make me happy>>
```

A link like `[[things that make me happy]]` would then link to that instead of other occurances of the text.

You can define keywords that will be automatically hyperlinked throughout the file by using radio targets. For example, if you're writing a jargon-filled document and you frequently need to refer to the definitions, it may help to make a glossary of terms such as "regexp" and "radio target", and then define them in a glossary section at the end of the file, like this:

```
*** glossary
<<<radio target>>> A phrase automatically hyperlinked whenever it appears.
<<<regexp>>> A regular expression. See the Emacs info manual.
```

Radio targets are automatically enabled when an Org file is opened in Emacs. If you've just added a radio target, enable it by moving the point to the anchor and pressing **C-c C-c** (`org-ctrl-c-ctrl-c`). This turns all instances of the text into hyperlinks that point to the radio target.

Project XXX: Publish Your Org Notes to HTML

You can keep your notes as a plain text file, or you can publish them as HTML or LaTeX. HTML seems to be the easiest way to let non-Emacs users read your notes, as a large text file without pretty colors can be hard to read.

To export a file to any of the formats that Org understands by default, type **C-c C-e** (`org-export`). You can then type **h** (`org-export-as-html`) to export it as HTML for websites. You can also type **l** (`org-export-as-latex`) to export it to LaTeX, a scientific typesetting language, which can then be published as PDF or PS. Another way to convert it to PDF is to export it as HTML, open it in OpenOffice.org Writer, and use the Export to PDF feature. You can also open HTML documents in other popular word processors to convert them to other supported formats.

By default, Org exports all the content in the current file. To limit it only to the visible content, use **C-c C-e v** (`org-export-visible`) followed by either **h** for HTML or **l** for LaTeX.

If you share your notes, you may want to export an HTML version every time you save an Org file. Here is a quick and dirty way to publish all Org files to HTML every time you save them:

```
(defun wicked/org-publish-files-maybe ()
  "Publish this file."
  (org-export-as-html-batch)
  nil)
(add-hook 'org-mode-hook  ❶
 (lambda ()
  (add-hook (make-local-variable 'after-save-hook) ❷
            'wicked/org-publish-files-maybe)))
```

❶ Every time a buffer is set to Org mode, this code adds wicked/org-publish-files-maybe to the ❷list of functions called after the file is saved.

What if most of your files are private, but you want to publish only a few of them? To control this, let's add a new special keyword `#+PUBLISH` to the beginning of the files that you want to be automatically published. Replace the previous code in your *~/.emacs* with this:

ch07-org-publish-files-maybe-2.el

```
(defun wicked/org-publish-files-maybe ()
  "Publish this file if it contains the #+PUBLISH keyword."
  (save-excursion
   (save-restriction
    (widen)
    (goto-char (point-min))
```

```
    (when (re-search-forward "^#+PUBLISH" nil t)
     (org-export-as-html-batch)
     nil))))

(add-hook 'org-mode-hook
 (lambda ()
  (add-hook (make-local-variable 'write-file-functions)
            'wicked/org-publish-files-maybe)))
```

and add `#+PUBLISH` on a line by itself to your *~/notes.org* file, like this:

```
#+PUBLISH
* Things that make me happy
```

When you save any Org file that contains the keyword, the corresponding HTML page will also be created. You can then use a program like rsync or scp to copy the file to a webserver, or you can copy it to a shared directory.

Project XXX: Integrate Remember with Org

You can use Remember to collect notes that you will later integrate into your outline. Add the following code to your *~/.emacs* to set it up:

```
(global-set-key (kbd "C-c r") 'remember)    ❶
(add-hook 'remember-mode-hook 'org-remember-apply-template) ❷
(setq org-remember-templates
      '((?n "* %U %?\n\n  %i\n  %a" "~/notes.org")))  ❸
(setq remember-annotation-functions '(org-remember-annotation)) ❹
(setq remember-handler-functions '(org-remember-handler)) ❺
```

This defines❶a handy keyboard shortcut for remember. It also configures Remember to❷use templates, and creates a template that ❸saves notes to *~/notes.org*. This also ❹creates org-compatible context links and ❺uses Org to save the notes.

With this code, you can type **C-c r n** (`remember`, notes) to pop up a buffer containing a link to the current file or buffer. Write your note, and type **C-c C-c** (`remember-buffer`) to save the note and close the buffer. You can use this to store snippets in your notes file, or to quickly capture an idea that comes up when you're doing something else.

Now you know how to sketch an outline, reorganize it, fill it in, brainstorm, stay motivated, and publish your notes. I look forward to reading what you have to share!

Daily Notes with Planner

In this section, you'll learn how to write a day-based journal using Planner, a personal information manager based on Muse. In addition to free-form notes on the page, you'll also be able to keep semi-structured notes typed in manually or captured using Remember. You'll also learn how to publish the resulting pages as HTML and RSS, and how to customize the output.

Project XXX: Set up Planner

If you haven't set up Planner yet, get the latest version of Planner, Muse, and Remember from the following sources:

- *http://www.mwolson.org/static/dist/planner-latest.tar.gz* or
- *http://www.mwolson.org/static/dist/planner-latest.zip*
- *http://www.mwolson.org/static/dist/muse-latest.tar.gz* or
- *http://www.mwolson.org/static/dist/muse-latest.zip*
- *http://www.mwolson.org/static/dist/remember-latest.tar.gz* or
- *http://www.mwolson.org/static/dist/remember-latest.zip*

Unpack the archives under your *~/elisp* directory and set up Planner by adding code like this to your *~/.emacs*:

ch7-planner-setup.el:

```
(add-to-list 'load-path "~/elisp/planner-latest")
(add-to-list 'load-path "~/elisp/muse-latest/lisp")
(add-to-list 'load-path "~/elisp/remember-latest")
(require 'planner)
(setq muse-project-alist
      '(("WikiPlanner"
       ("~/plans"   ❶ ; Or wherever you want your planner files to be
        :default "index"
        :major-mode planner-mode
        :visit-link planner-visit-link)
       (:base "planner-xhtml"
        :path "~/public_html/plans")))) ❷
(global-set-key (kbd "C-c p") 'planner-goto-today) ❸
(global-set-key (kbd "C-c P") 'planner-goto) ❹

(global-set-key (kbd "<f9> p") 'planner-goto-today) ❺
```

```
(global-set-key (kbd "<f9> P") 'planner-goto) ❻
(global-set-key (kbd "<f9> <f9>") 'planner-goto-today) ❼
(global-set-key (kbd "<f9> <f8>") 'planner-goto-previous-daily-page)
(global-set-key (kbd "<f9> <f10>") 'planner-goto-next-daily-page)
```

This defines a project that uses❶*~/plans* as the directory for Planner files (also known as your planner directory) and❷*~/public_html/plans* as the directory for published Planner files. Create these directories if they don't exist, or change them to directories you want to use.

You'll also want to define some convenient keyboard shortcuts. Make it easy to review today's notes and a particular day's notes with ❸**C-c p** for `planner-goto-today` and ❹**C-c P** for planner-goto. I like using a set of keyboard shortcuts based on a function key in addition to the **C-c** shortcuts, and ❺ **F9 p** and ❻ **F9 P** are handy. You can quickly flip through pages with ❼ **F9 F9** to jump to today's page, **F9 F8** to jump to the previous page, and **F9 F10** to jump to the next page.

Now you're ready to use Planner for quick daily notes. Open today's page with **F9 F9** or **C-c p** (`planner-goto-today`). By default, you'll see the following template:

```
 * Tasks

 * Schedule

 * Notes
```

You can stop here and just use Planner as a convenient way to create date-stamped notes. Type in whatever notes you want in whatever format you want. When you save your notes, they will be stored as plain text files in `~/plans` with filenames of the form yyyy.mm.dd (*year.month.day*).

You can open files using **C-x C-f** (`find-file`), and Emacs will automatically open them in Planner mode. You can also use **C-c P** or **F9 P** (`planner-goto`) to jump to a specific date, which you can specify as *yyyy.mm.dd* (example: `2008.02.01`), *mm.dd* (example: `2.1`), or just *dd* (example: `1`). When you call `planner-goto` interactively, Emacs displays a calendar. You can choose dates using the mouse or using keyboard navigation. Press **RET** (`planner-calendar-select`) in the Calendar buffer to select a date.

Using more structure in your Planner notes will allow you to do useful things like capturing notes from anywhere in Emacs (Project XXX: Remembering to Planner) or publishing notes to RSS (Project XXX: Publish Your Planner as RSS). Here's what a Planner note looks like:

```
* Tasks

* Schedule

* Notes

.#1 This is a note headline

This is the note body.

It includes everything until the next heading (like * Tasks) or headline.

.#2 This is another headline

And so on.
```

To easily capture timestamped, hyperlinked notes to Planner, read the next project and set up Remember for Planner.

Project XXX: Remember to Planner

Remember is a quick way to add timestamped notes to your Planner file, and you can hook into it to publish your notes as HTML or RSS. Here's how to configure it for Planner:

```
(require 'remember)
(require 'remember-planner)
(global-set-key (kbd "C-c r") 'remember)
(global-set-key (kbd "<f9> r") 'remember)
(setq remember-annotation-functions planner-annotation-functions)
(setq remember-handler-functions '(remember-planner-append))
```

Use **C-c r** (`remember`) to pop up a buffer for your notes, and use **C-c C-c** (`remember-buffer`) to save the note. You will be prompted for a plan page to which to save the note. Press RET to accept the default of saving the note only on today's planner page.

Now that you've taken notes, how can you review them?

Project XXX: Review Your Notes in Emacs

You can view today's notes with **M-x planner-goto-today**. **M-x planner-goto** lets you jump to a specific date, and **M-x planner-goto-yesterday** and **M-x planner-goto-tomorrow** let you flip back and forth. These are handy functions, so here are some sets of shortcuts based on the configuration suggested in "Project XXX: Set up Planner":

Show the page for today	planner-goto-today	F9 F9	C-c C-j C-j	C-c p
Jump to a given day	planner-goto	F9 P	C-c C-j C-d	C-c P
Jump to the previous day	planner-goto-previous-daily-page	F9 F8	C-c C-j C-p	
Jump to the next day	planner-goto-next-daily-page	F9 F10	C-c C-j C-n	

Table 3: Keyboard shortcuts for navigating day pages

C-c C-j C-y (planner-goto-yesterday) and **C-c C-j** C-t (planner-goto-tomorrow) are useful when you want to move by day. Use these keyboard shortcuts to quickly flip through pages. If you write daily or weekly notes, you can use **F9 F8** (planner-goto-previous-daily-page) to scan through your notes.

There are two built-in functions for searching your notes. **M-x planner-search-notes** searches titles for a regular expression and displays all of the matching notes in a single buffer, with hyperlinks back to the files. **M-x planner-search-notes-with-body** searches the full text of the note. These functions search each file individually without any optimizations. If you have hundreds of Planner files and would like a faster way to search, you can use the UNIX command-line tool grep, or a search tool such as Google Desktop. Because Planner notes are saved in plain text files, you can search them easily.

You can view a quick summary of Planner day pages with planner-notes-index.el. Add the following to your *~/.emacs* to load it:

```
(require 'planner-notes-index)
```

You can then call **M-x planner-notes-index** to see an index of all the note headlines, like this:

```
* 2008

** 2008.01

*** 2008.01.29

 - This is another test 21:38
 - This is a test

*** 2008.01.20
```

```
  - This is a test 21:48
  - Hello, world!  19:30
```

You can also limit the displayed entries using `planner-notes-index-days`, `planner-notes-index-weeks`, `planner-notes-index-months`, and `planner-notes-index-years`. Each function prompts for a number and displays the note headlines from the past *N* days, weeks, months, or years.

If you have a lot of notes, you may be interested in sharing your notes with others. If you share your Planner files, people can search through your Planner project using the same functions you do. However, you will probably want to share your Planner files with non-Emacs users who don't like reading plain text. How can you publish it in a friendlier format?

Project XXX: Publish Your Planner Notes as HTML

Planner is built on Muse, a markup engine that makes it easy to publish notes in a variety of formats. HTML is one of the most popular formats. You can publish the current file as HTML with **M-x muse-publish-this-file**, which prompts you for a style (example: **planner-html**) and a directory (example: *~/public_html/plans*). The HTML file will be created in that directory.

You can set publishing defaults by using *muse-project-alist*. In the basic configuration, you added this to your *~/.emacs*:

```
(setq muse-project-alist
      '(("WikiPlanner"
       ("~/plans"   ; Or wherever you want your planner files to be
        :default "index"
        :major-mode planner-mode
        :visit-link planner-visit-link)
       (:base "planner-xhtml"        ❶
        :path "~/public_html/plans"))))  ❷
```

This specifies ❶planner-xhtml as the base style, and ❷~/public_html/plans as the publishing directory. You can then call **M-x muse-project-publish-this-file** to publish the current Planner page to your output directory, and use a program like ftp, scp or rsync to copy the file to your webserver.

What if you want to automatically publish whenever you save a day page? Again, Emacs' extensibility comes to the rescue. Add the following code to your *~/.emacs*:

ch07-muse-publish-this-page:

```
(add-hook 'planner-mode-hook ❶
          (lambda ()
            (add-hook 'after-save-hook
                      'wicked/muse-publish-this-page))) ❷
(defun wicked/muse-publish-this-page ()
  "Save and publish this page if it is not a private page."
  (interactive)
  (unless (or muse-publishing-p   ❸
              (muse-project-private-p (buffer-file-name))) ❹
    (let ((new (not (muse-project-page-file (planner-current-file)
                                            planner-project))))
      (save-buffer)
      (when new (muse-project-file-alist nil t)) ❺
      (muse-project-publish-this-file)))) ❻
```

This sets up ❶ all newly-created Planner mode buffers to ❷ automatically publish their contents whenever they are saved. When Emacs is doing so, it is important to avoid ❸ getting into an infinite loop or ❹ publishing private information. It's also a good idea to refresh the list of project files so that new pages are noticed. Once that's done, Emacs will go ahead and ❻ publish the file.

If you publish your notes, you probably want to specify private notes. How?

Project XXX: Keep Private Notes

One way to keep notes private is to use private pages. By default, muse-project-private-p indicates that a file is private if other people do not have read permissions for it. This works only under UNIX and Linux, where you can use `chmod o-r` *filename* to remove the read permissions for other people. Besides, you often don't want to make the entire page private. Protecting just a small section of your daily notes is usually enough.

You can define a custom tag <private>...</private>, which removes everything within the tags. Add the following code to your *~/.emacs*:

ch07-planner-private-tag:

```
(defun wicked/planner-private-tag (beg end &optional attrs)
  "Delete everything within the <private> and </private> tags."
  (interactive)
  (let ((inhibit-read-only t))
    (delete-region beg end))  ❶
  (planner-insert-markup "..."))   ❷
```

```
(add-to-list 'muse-publish-markup-tags    ❷
             '("private" t nil nil wicked/planner-private-tag))
```

This defines a function that ❶ replaces all the text between the start and end tag ❷ with "...". This function is then ❷ added to the list of tags that muse-publish recognizes. The enclosed text will be in your Planner page, but not in any published versions such as HTML or RSS. Confirm that this is the case before uploading it, as any mistakes in typing <private> (for example, <prviate>) will end up in exposed information.

If you want your notes protected even in your Planner pages, you can encrypt the sensitive information. One way to protect it from casual reading is to use ROT13, which turns all As into Ns, Bs into Ns, and so forth. For example, "The quick brown fox jumps over the lazy dog" encrypted using ROT13, is "Gur dhvpx oebja sbk whzcf bire gur ynml qbt". If you apply ROT13 another time, you will get the original text. It's not at all secure, but it may stump casual readers.

To encrypt some text, select the region and call **M-x rot13-region**. This replaces the region with the ROT13 equivalent. To decrypt it, select the text and call **M-x rot13-region** again.

A stronger way to protect your notes is to use the GNU Privacy Guard (`GnuPG or GPG`) and pgg.el, the Emacs interface to GPG. Properly setting up a public and private key is beyond the scope of this book, but you can find information at *http://www.gnupg.org* . First, set up GPG.

In Linux and UNIX systems:

1. Install GNU Privacy Guard (gnupg on most distributions).
2. Use **gpg --gen-key** at the command line to generate a secret key and a public key.
3. Choose a hard-to-guess passphrase.

In Microsoft Windows:

1. Download and install GnuPG for Windows from *http://www.gpg4win.org*.
2. Start GnuPG For Windows - GPA.
3. Generate a key.
4. Choose a hard-to-guess passphrase.

Once you've set up GPG, you can use it to encrypt notes in Emacs. Select the text you want to encrypt, type **M-x pgg-encrypt-region**, and type in the name you specified for your public key. To decrypt text, select the text, type **M-x pgg-decrypt-region**, and type in your passphrase. Keep your secret key and your passphrase secret, and you should be fine.

If you get "Search failed" results and you're on Microsoft Windows, then the *pgg.el* that comes with your Emacs does not understand the DOS-style linebreaks used in Microsoft Windows. You will also need to add the following code to your *~/.emacs*:

ch7-pgg-fix.el:

```
(require 'pgg)
(defmacro pgg-save-coding-system (start end &rest body)
  `(if (interactive-p)
       (let ((buffer (current-buffer)))
       (with-temp-buffer
         (let (buffer-undo-list)
           (insert-buffer-substring buffer ,start ,end)
           (prog1 (save-excursion ,@body)
             (push nil buffer-undo-list)
             (ignore-errors (undo))))))
     (save-restriction
       (narrow-to-region ,start ,end)
       ,@body)))
```

This removes the encoding code so that DOS-style linebreaks won't confuse *pgg.el*. Evaluate that code and try again. If it still doesn't work, consider switching to Linux, as most Emacs modules are developed and tested on Linux or UNIX systems.

Project XXX: Publish a Planner Index

You can make your Planner easier to navigate with a calendar that shows which days have notes. To do this, call **M-x planner-index**, and save the generated file to a Planner page like WikiIndex. You can then use **M-x muse-publish-this-file** to convert that page to HTML. To make this a one-step process, create the following function:

```
(defun wicked/planner-publish-index ()
  "Publish an index of the plan pages."
  (interactive)
  (save-window-excursion
    (planner-find-file "WikiIndex")
    (erase-buffer)
    (insert (planner-index-as-string t t))
    (save-buffer)
    (muse-project-publish-this-file)))
```

You can then call **wicked/planner-publish-index** to update the index.

Project XXX: Publish Your Planner Notes as RSS

If you're publishing your daily notes, then you probably want to provide a feed so that people can subscribe to your notes. Really Simple Syndication (RSS) is a popular standard for making blogs, news, and other information available through different readers. In this project, you will learn how to configure Planner's built-in RSS support.

Add this to your *~/.emacs* to publish notes to an RSS file called *~/public_html/blog/blog.rdf*:

```
(require 'planner-rss)
(setq planner-rss-category-feeds
      '(("."   ❶
         "~/public_html/blog/blog.rdf"   ❷
         "<?xml version=\"1.0\"?><rss version=\"2.0\"><channel> ❸
<title>Title of Blog</title>  ❹
<link>http://example.com/blog</link> ❺
<description>Description of Blog</description> ❻
</channel></rss>")))
(setq planner-rss-base-url "http://example.com/blog/")  ❼
(setq planner-rss-feed-limits '(("." nil 10))) ❽
```

`planner-rss-category-feeds` ❶ specifies which posts match this category, and this can be a regular expression that matches the text of the Planner note or a function that returns non-nil if the post matches certain criteria. In this case, `.` means match all entries.

The second element in the list ❷ means all entries should be copied into a file called *~/public_html/blog/blog.rdf*. In a Linux or Unix environment with a webserver set up for user websites, *~/public_html* is usually the directory for a user's personal website. Create the directory before publishing to RSS, or you might get a message like `Use M-x make-directory RET RET to create the directory and its parents`. If your website is hosted on a different computer, change ❷ to a file that can be conveniently copied to your webserver using a program like rsync, scp, or ftp.

If the file specified ❷ does not yet exist, then it is initialized with ❸ blank RSS file ❹ with the title of your blog, ❺ with a link to your blog's homepage, and ❻ with a short description of who you are and what you write about. If the ❷ specified file already exists, then you can simplify this assignment into something like this:

```
(setq planner-rss-category-feeds
      '(("."
         "~/public_html/blog/blog.rdf"
         "")))
```

Because the RSS feed points to your original entries, you will need to specify the Web location of ❼your Planner publishing directory. For example, I publish my Planner pages to *http://sachachua.com/notebook/wiki/*, so that would be my `planner-rss-base-url`. That way, planner-rss knows how to link to your published files.

By default, planner-rss does not limit the size of your published RSS file. However, many RSS readers and services have a hard time operating on large RSS feeds. For example, you can ❽ limit all the RSS feeds produced by planner-rss to a maximum of 10 entries. The first element is a regular expression matching the filename of the RSS file, the second element is the total number of characters allowed for the entire file, and the third element is the total number of entries allowed. `nil` means unlimited. You can specify both limits like this:

```
(setq planner-rss-feed-limits '(("." 50000 20)))
```

This limits it to 50,000 characters or 20 entries, whichever is smaller.

To manually publish notes to your Planner, move your point to somewhere within the note on your daily page. For example, on your Planner page for 2008.01.01, you might have

```
.#1 Hello, world!  19:30

This is a test post.
```

Move your point to that note and use **M-x planner-rss-add-note**. Check your *~/public_html/blog/blog.rdf* to see if the item has appeared. You can publish the note to a different RSS feed by using a prefix argument like this: **C-u M-x planner-rss-add-note**.

If you use Remember to save notes to your Planner day page, you can set up Planner to automatically publish those notes to your feed. Add the following to your *~/.emacs*:

```
(add-hook 'remember-planner-append-hook 'planner-rss-add-note t)
```

Project XXX: Publish a Note to More than One Feed

You may want to offer more than one feed to your website visitors. For example, you may have an Emacs-related feed that can be included in Planet Emacsen (*http://planet.emacsen.org*), or a weekly summary feed that people can check once in a while.

You can publish a note to a specific feed. Make sure that you've initialized the feed by creating a file that contains something like this:

```
<?xml version="1.0"?><rss version="2.0"><channel>
<title>Title of Blog</title>      ❶
<link>http://example.com/blog</link>    ❷
<description>Description of Blog</description> ❸
</channel></rss>
```

Fill in the template with ❶your blog name, ❷a link to your blog homepage, and ❸a description of the feed.

After you've initialized the file, move your point to the note you want to publish and type **C-u M-x planner-rss-add-note**. This will prompt you for a filename. Specify the feed that you want to publish to, and planner-rss will publish the note to it. This observes the limits set in *planner-rss-feed-limits*.

To automatically publish your notes to a number of different feeds, add more entries to planner-rss-category-feeds. *planner-rss-category-feeds* controls the behavior of `planner-rss-add-note`, so it applies whether you call **M-x planner-rss-add-note** manually or you use the *remember-mode-hook* suggested in the previous project. Notes are copied to all files that match the criteria, which can be a regular expression or a function that takes the marked-up HTML of the note as an argument. For example, to copy all posts containing `emacs` to *emacs.rdf*, replace your *planner-rss-category-feeds* setting in *~/.emacs* with this:

```
(setq planner-rss-category-feeds
      '(("emacs" "~/public_html/blog/emacs.rdf" "")
        ("." "~/public_html/blog/blog.rdf" "")))
```

You can use functions for more sophisticated criteria. For example, if you want to share the recipes you've tried but you don't want to share the ones that sucked, you could use a setting like this:

```
(setq planner-rss-category-feeds
      '(((lambda (text)
           (and (string-match "recipe" text)
                (not (string-match "sucked" text))))
```

```
          "~/public_html/blog/recipes.rdf" "")
        ("emacs" "~/public_html/blog/emacs.rdf" "")
        ("." "~/public_html/blog/blog.rdf" "")))
```

The function should take a single argument (the marked-up text) and return non-nil if this note should be copied to the specified feed. Consistently mention "recipe" somewhere in your cooking posts and "sucked" somewhere in the posts about your failures, and Emacs will take care of sorting out the rest.

Project XXX: Update a Note Published to RSS

planner-rss is an excellent way to sharpen your eyesight, as you will undoubtedly discover embarrassing errors right after a note is published to several dozen fields. Because planner-rss makes a copy of your note, any changes you make to your original Planner page are not automatically reflected in your RSS feeds. If you call **M-x planner-rss-add-note** again, you will have duplicate items. One way to correct your post is to manually edit your RSS feed after you update your Planner page, but that wouldn't be the Emacs solution. The Emacs solution, of course, is to define a function to do most of the work for you.

Here are a set of functions that delete the incorrect note and re-create it. They work in simple cases, such as when you've made a small typo or thought of something to add. Feel free to modify this if you need something more powerful.

ch07-planner-rss-update.el:

```
(defun wicked/planner-update-note ()
  "Update this note in RSS and Planner."
  (interactive)
  (let ((inhibit-read-only t))
    (wicked/planner-rss-undo-this-note)   ❶
    (planner-update-note)                 ❷
    (planner-rss-add-note)))              ❸

(defun wicked/planner-rss-undo-this-note ()
  "Delete the current entry from the RSS feeds it matched."
  (interactive)
  (save-excursion
    (save-restriction
      (planner-narrow-to-note)
      (let* ((feeds planner-rss-category-feeds)
```

```
            (info (planner-current-note-info))
            (page  ❹
             (concat "<link>"
                     planner-rss-base-url
                 (muse-page-name)
                 ".html#"
                     (planner-note-anchor info)
                     "</link>"))
            files)
        (while feeds ❺
        (let ((file (car (cdr (car feeds)))))
          (with-current-buffer (find-file-noselect file)
            (goto-char (point-min))
            (when (re-search-forward page nil t)
            (wicked/rss-delete-item)
            (save-buffer)))
          (setq feeds (cdr feeds))))))))

(defun wicked/rss-delete-item ()
  "Delete the current RSS item."
  (interactive)
  (let ((inhibit-read-only t))
    (delete-region
     (if (looking-at "<item>")
         (point)
       (when (re-search-backward "<item>" nil t)
         (match-beginning 0)))
     (when (re-search-forward "</item>" nil t)
       (match-end 0)))))
```

Call **M-x wicked/planner-update-note** from your changed note, and as long as the page name and the note number haven't changed, the RSS file should be updated. `wicked/planner-rss-undo-this-note` ❶ deletes the existing note from the RSS feed. `planner-update-note` ❷ updates linked Planner pages, and `planner-rss-add-note` ❸ publishes the note to the RSS feed again.

Here's how the corresponding note is deleted from the feeds. `wicked/planner-rss-undo-this-note`❹ identifies the note by the published filename and the note number, ❺ loops over the feeds listed in *planner-rss-category-feeds* and deletes the item whenever found.

Use **M-x wicked/planner-update-note** after you change a note on your Planner page, and your RSS feeds will be updated. Copy them to your webserver, and your subscribers should see the new version the next time they check. If systems like Feedwordpress don't recognize the update because the publication date of the item is the same as the original, change the timestamp in the file or force a refresh on the system.

Blogging From Emacs

In this section, you'll learn how to blog from Emacs. If you want to keep all your notes on your computer and you don't need to provide an RSS feed or commenting facilities, follow the instructions on "Project XXX: Publish Your Org Notes to HTML" or "Project XXX: Publish Your Planner Notes to HTML" and "Project XXX: Publish Your Planner Notes as RSS". If you want to make it more like a blog, you have several options:

- Using Emacs as an external editor for your browser
- Posting from Emacs directly to a blog
- Syndicating an RSS feed of Planner notes into another blog

If you already have an existing blog and you want to occasionally use Emacs to edit posts, you can integrate Emacs into Mozilla Firefox and use it to edit text areas. Read "Project XXX: Edit Text in Emacs from Mozilla Firefox". If you want to keep things simple, read "Project XXX: Post directly to a blog" and find out if your blog is supported.

If you want to keep notes on your computer but provide a more sophisticated blog platform such as Wordpress, follow the instructions in "Project XXX: Publish Your Planner Notes as RSS" to publish an RSS feed and the instructions in "Project XXX: Syndicate Planner RSS into Wordpress" to import it into your other blog.

Sacha here! Most of the integration examples here are about Wordpress.org because I like Wordpress and that's what I have on my webserver. Blosxom and pyblosxom are also popular among the Emacs crowd. Hack it to fit.

Project XXX: Edit Text in Emacs from Mozilla Firefox

If you spend most of your time in a Web browser, you'll probably like Mozilla Firefox, which is almost as extensible as Emacs. There are several extensions that will allow you to use Emacs as an external editor for editing text areas, viewing source code, or even sending mail. Mozex has more features than It's All Text, and the development version of Mozex works with Mozilla Firefox 2.0 and later versions.

Add the following line to the end of your *~/.emacs* in order to allow your programs to connect to your current Emacs:

```
(server-start)
```

Install Mozex from *http://mozex.mozdev.org/development.html*. After you restart Mozilla Firefox, right-click on a page and choose **Mozex - Configuration** from the context menu. On the Textarea tab, set your text editor to the full path to emacsclient, adding -a emacs %t to the end.

Here are some example commands:

- Linux and UNIX: `/usr/bin/emacsclient -a emacs %t`
- Microsoft Windows: `c:\emacs\bin\emacsclient.exe -a emacs %t`

This attempts to connect to Emacs to edit a temporary file, and if a connection can't be established, it starts a new Emacs and opens the temporary file.

If you get a "cannot run executable" error, double-check your command. It needs to include the full path to the command as well as any extensions. For example, on Microsoft Windows, you need to specify the *.EXE* extension as well.

You can now use Mozex to edit text areas. Mozex works only on plain text areas, not rich text editors that allow you to make text bold and italicized inside the browsers. Switch to the code or HTML view to get a plain text area that Mozex can work with, then switch to the rich text editor to format your work.

When you use Mozex to edit a text area, emacsclient opens a temporary file in Emacs. After you finish editing the file, save the file and type **C-x #** (`server-edit`) to mark it done. Mozilla Firefox will show your updates.

If you spend more time in Emacs than in a web browser, you might consider posting directly to your blog from Emacs instead of opening a web browser when you want to post. Here's how.

Project XXX: Post Directly to a Blog

Find out if your blog will accept entries through e-mail. If so, then all you need to do is set up Emacs to send e-mail (see chapter XXX). For example, Blogger.com and Livejournal.com allow you to post by e-mail. Wordpress.com-hosted blogs do not currently accept posts by e-mail, but if you host a Wordpress blog on your own server, you can configure that feature by following the instructions at *http://codex.wordpress.org/Blog_by_Email* .

Posting from Emacs through e-mail works well for plain text. To post HTML, you might want to use a blogging client. For example, you can use *weblogger.el* and *xml-rpc.el* to post to blogs supporting

Blogger, MetaWeblog, and Movable Type application programming interfaces (APIs). Wordpress supports the Blogger, MetaWeblog, and Movable Type APIs, so you can use Weblogger to post entries directly to your blog.

You can get Weblogger from *http://www.emacswiki.org/cgi-bin/wiki/weblogger.el* and XML-RPC for Emacs from *http://www.emacswiki.org/cgi-bin/wiki/xml-rpc.el*, or you can check out the source code from *http://savannah.nongnu.org/projects/emacsweblogs*.

Download *weblogger.el* and *xml-rpc.el* to your *~/elisp* directory and add the following code to your *~/.emacs*:

```
(add-to-list 'load-path "~/elisp")    ❶
(require 'weblogger)
(global-set-key (kbd "C-c b s") 'weblogger-start-entry)  ❷
```

Then call **M-x weblogger-setup-weblog** to configure your weblog. Emacs will prompt you for the URL to the XML-RPC interface for your blog. If you don't know the URL, give your blog address and Emacs will try to autodetect the settings. It will then ask you for your username and password, and the name of this configuration. Yo can call **M-x weblogger-setup-weblog** again to set up other weblogs.

To start an entry, type **C-c b s** (`weblogger-start-entry`). This displays a buffer like this:

```
Subject:
From: your-user-name
Newsgroup: your blog title
--text follows this line--
```

Move the point to the first line and type in a subject. Type your text below `--text-follows-this-line--`. Depending on the configuration of your blog on the server, linebreaks may be automatically converted to
 tags so that they will appear in the blog post. Check this with a test post.

After you type your entry, use **C-x C-s** (`weblogger-publish-entry`) to publish the entry to the weblog. To publish it to a different weblog, use **C-u C-x C-s** (`weblogger-publish-entry with a prefix argument`) and specify the name of the blog configuration you want to use. To save a post as a draft but not publish it, type **C-c C-c** (`weblogger-send-entry`).

You can also change the default weblog by typing **M-x weblogger-select-configuration** and the name you specified when you configured the blog. Although the blog entry screen will not be updated, if you publish the entry, it will use the configuration you selected. If you want to update the blog entry screen so that you can confirm the selection of the blog, type **C-c C-w** (`weblogger-update-weblog`). This updates the title of the blog in the Newsgroup: header.

Weblogger allows you to specify the categories of posts for blogs that support the MetaWeblog API. The API standard limits categories to only the categories that already exist on the blog. Categories that do not exist will be silently ignored, and no new categories will be created. You can manually specify categories by typing in the following header:

```
Subject: Hello, world!
From: your-user-name
Newsgroup: your blog title
Keywords: emacs, test
--text follows this line--
The quick brown fox jumps over the lazy dog.
```

In this example, the post will be assigned to two categories (`emacs and test`) if they exist.

You can configure weblogger to automatically insert the Keywords: header when you start a new entry by adding the following code to your `~/.emacs`:

ch07-weblogger-add-keywords-header.el:

```
(defun wicked/weblogger-add-keywords-header ()
  "Add a Keywords: header if there isn't one yet."
  (if (message-fetch-field "Keywords")
      (message-goto-body)
    (message-position-on-field  "Keywords")))
(add-hook 'weblogger-start-edit-entry-hook
          'wicked/weblogger-add-keywords-header)
```

You can use **M-x weblogger-fetch-entries** to retrieve and edit blog posts. The number of entries is controlled by weblogger-max-entries-in-ring, which is set to 20 by default. You can then use **C-c C-p** (`weblogger-prev-entry`) and **C-c C-n** (`weblogger-next-entry`) to go through the entries. If you change a post and then use those navigation commands to view another post, your changes will be

uploaded to the server and saved as a draft. You can publish your changes with **C-x C-s** (`weblogger-save-entry`) or delete the current entry with **C-c C-k** (`weblogger-delete-entry`).

Weblogger and XML-RPC for Emacs do not implement all of the features you might find in your blog administration interface, but they do allow you to quickly post to a number of blogs. Another way to publish to your blog is to use Planner to create an RSS feed and syndicate it into another blogging platform such as Wordpress.

Project XXX: Syndicate Planner-RSS into Wordpress

Planner is a powerful, flexible way to write your blog posts, but adding blog features such as comments and categories can require some hacking. Get the best of both worlds by publishing your post using Planner and syndicating it into another blogging platform such as Wordpress.

To integrate Planner and Wordpress, follow the steps in "Project XXX: Publish Your Planner Notes as RSS". Upload the resulting RSS feed to a webserver. Install Wordpress on your server following the instructions at *http://www.wordpress.org*. Then install either of the following plugins:

- WP-o-Matic (*http://devthought.com/wp-o-matic-the-wordpress-rss-agreggator/*)
- Feedwordpress (*http://wordpress.org/extend/plugins/feedwordpress/*).

Configure this plugin with your RSS feed, set up syndication, and your Planner blog will be republished with a slicker interface.

Project XXX: Microblog with Twitter

You can also post quick notes to Twitter (*http://www.twitter.com*). Download *twit.el* from *http://www.emacswiki.org/cgi-bin/emacs/twit.el* and add it to your *~/elisp*. Add the following to your *~/.emacs* to load it:

```
(add-to-list 'load-path "~/elisp")
(require 'twit)
(global-set-key (kbd "C-c b t") 'twit-post)
```

This binds `twit-post` to **C-c b** (for blog) **t** (for twitter). Twit will prompt you for your username and password the first time that you use it in an Emacs session, and it will save it for the rest of the session.

Wrapping Up

In this chapter, you learned how to capture notes using Remember, work with outlines using Org, and keep a day-based journal using Planner. You also learned how to post to other blogging systems.

The next chapter on managing your tasks goes into more detail about how to use Org and Planner. Read on!

END OF CHAPTER

Author's notes (for people who are reading this on the Internet)

Hi, I'm Sacha Chua (sacha@sachachua.com). I'm writing a book called Wicked Cool Emacs about a program that isn't just a text editor but a way of life. You can find out more about it at http://sachachua.com/wp/category/wickedcoolemacs . I'd love to hear what you think of this draft and how we can make this a better book!