A Style Guide for GNU Documentation

by Ron Hale-Evans

Based largely on comments by Robert J. Chassell
and Richard M. Stallman

1 Basic points of style

The goal of free GNU documentation is to help the users and developers of GNU software. There is no need for you to provide examples for software running under other operating systems. In particular, there is no need for you to provide examples for operating systems that take away your freedom.

- Show, don't just tell. Use plenty of examples (but don't be overly redundant).
- Move slowly. Do not impose too much of a cognitive load at once on the reader.
- Explain things in detail if you are writing a tutorial for a novice. Since one tends to under-explain anyway, pretend you are writing for an intelligent person who lives in an undeveloped country and is unfamiliar with the technology you are explaining.
- Don't say too little. If you cannot include enough information on a topic to do more than tantalize a novice, omit it entirely.
- Do not assume the reader has specific knowledge of mathematics or computer science when it is possible she doesn't. You may have to explain what an integer is or what a byte is, at least at the level of a tutorial.
- Explain your value judgments. For example, if you say a code snippet is or is not "useful", explain *why* it is or is not. Value judgments can only be formed by people with knowledge of the relevant subject, and if the reader had the knowledge you use to form your judgments, she probably wouldn't need to read your documentation!
- If necessary, repeat yourself, especially if the information you are repeating is important and might be missed the first time. Also, if your reader is unlikely to remember a minor point that is nevertheless important to understand a major one, it is acceptable to repeat the information.
- Avoid editorializing, either about things outside the text ("As we know, every operating system but GNU sucks"), or about the text itself ("At last, we can discuss. . .").
- Design your text for a blind person; this is a good discipline. Documents, especially web pages, turn out much better. When you want to know how a document will sound to a blind person, you can run it through Festival or Emacspeak.
- Diagrams are sometimes helpful. Similarly, tables and lists of categories (variable types, types of operator, etc.) can help the reader encompass a large amount of information without a lot of superfluous connective text.
- Think of problems the reader might encounter — "gotchas" that you might have experienced yourself — then point them out. For example, in C and C++, point out the difference between `=` and `==`.

- Explain conventions. Note that software programming and usage often relies on conventions that are not obvious. For example, a '`0`' return code in a C program signifies "zero errors". It is good to explain a convention such as this.
- Always tell people how to pronounce code when you introduce new terms. For example, if you are explaining pointers, tell the reader that `*my_ptr` can be pronounced "the contents of the memory location `my_ptr`." The idea is to teach those who sound things out when reading to pronounce code the right way, rather than to come up with an idiosyncratic, personal method of reading, which can hurt their ability to learn the language.

  People who do not pronounce words, but depend entirely upon visualization, will not care much for this, but will not be hurt. Indeed, they will benefit, since they need to learn pronunciation in order to talk with other programmers.
- Qualify your statements. Don't simply say, for example, "Parameters must have their types declared." Must *all* parameters have their types declared? If so, say so; if not, state which parameters must have their types declared and which must not; and give examples where necessary.

2 Ordering your text

Write about first things first.

- Write about the most important things in a section first. You may want to give each its own subsection. Don't make the mistake of writing, "Blah, blah, and blah. Oh, and by the way, this is really important: . . ."
- Put important notes to the reader in subsections of their own. This tells the reader the notes really are important.

  While "first things first" usually applies, in some cases, the very end of a section is the best place for an important note, perhaps prefixed with '`@strong{Important:}`'. People tend to remember best the things they are shown first and last. Also, an important note can sometimes tie up a section very nicely.
- Order the information in your nodes from simple to complex.
- Don't use terms without defining them, at least in a brief, preliminary way. Do not use them *in the process of* defining them. Here, for example, is a classic error: '`This variable can take only @dfn{Boolean values}: true and false.`'
- Make your assumptions clear *before* you use them. For example, if you assume that the reader knows basic trigonometry, say so before you launch into an example involving it. You might also give pointers to where the reader could *learn* about the subject in question.
- Recursion and nested data structures are difficult. Your text can easily get out of control. Be extra careful to phrase your explanations clearly so the reader does not end up in a tangle.

  Bad: "All variables local to a block are invisible outside their block, but visible within every block their block contains."

  Good: "A local variable is visible within its own block and the ones that block contains, but invisible outside its own block."
- If your chapter is titled "About foo and bar", do not discuss your topics in the order "bar" and "foo". Be parallel and consistent throughout the section in question. This may mean you will have to order your text carefully in advance, but your readers will thank you.
- If you have two tables or lists of information that discuss the same items, combine them! Don't make the reader flip from one to the other, correlating them in her head.
- Don't combine different topics in the same paragraph or node. If you want to start a new topic, start a new paragraph or a new node.
- After you have made an important point in a paragraph, end the paragraph and let the reader "walk away with" that information. Don't clutter the paragraph with details, trailing off into irrelevance; save the details for later.

- When you are explaining a feature of a program, it is often helpful to awaken the reader's interest by first outlining a problem the feature solves or a need it fulfills. Write text that "motivates" the reader to understand why the feature is needed. You should assume that most people will not themselves think that they need the feature ahead of time, and that when the feature is introduced, only the really smart readers will figure out for themselves why it is a good idea.

3 Code examples

Examples should follow the GNU style. Consult the *GNU Coding Standards* for further information.

- Give sample output for code examples wherever possible.
- Don't waste the reader's time with frivolous examples that have no real use. For example, in the *GNU C Tutorial*, it was judged too frivolous to show the reader how to print out the values of pointers (of the pointers themselves, not the addresses pointed to), even though earlier editions of the book had done so.
- When you discuss a function, do not include the parentheses in its name unless you are illustrating a function call. For example, use `cos` rather than `cos()`.
- In an example, snuggle code up to the `@example` and `@end example` lines; do not insert blank lines between the lines containing the formatting commands and the lines containing the code.
- Always check your code examples by compiling and running them before including them in your text. This applies even to small examples. Double-check your mathematical examples as well as your code. Nothing will make your reader lose confidence in your documentation faster than catching you in a simple error.
- Use the present "timeless" tense when talking about what a code example does. Example: "The `foo` function takes an integer variable `bar` and multiplies it by 5."
- Put ellipses inside dummy code blocks, unless you want to imply they are no-ops.
- In examples of code, use all-caps only for macros and the like that are normally written in uppercase letters. Use lowercase letters for everything else.
- Don't use examples that will become dated. You don't know how long your text will be read. Example: If you are writing in the year 2002, and you want to use an example of a variable containing a year, rather than creating a variable called `cur_year` and making it equal to '`2002`', it is better to create a variable called `moon_landing` and make it equal to '`1969`'.
- In your code examples, use variable names that are concrete rather than abstract. Concrete names are less confusing. For example, a variable called `cost_of_lunch` is better than one called `humdinger` or `foo`. On the other hand, do not use variable names that are so concrete that the example itself takes over and the lesson it is supposed to convey is lost.
- Satisfy the reader's curiosity about whether alternate coding practices are possible, but make your recommendations clear.

4 Metaphors

People reason using metaphors.

- Develop your metaphors explicitly. For example, if you say local variables are "invisible" outside their functions, explain that this usage stems from a metaphor in which functions are something like buildings and local variables are like people looking from one building to another.
- Jargon often has a metaphorical underpinning. For example, pointers "point to" memory addresses. It is helpful to explain these metaphorical underpinnings when introducing a jargon term.
- Explain where your metaphors fail. For example, when explaining pointers in C, explain that while, with the same finger, you can point to anything you like in real life (whether it be animal, vegetable, or mineral) a given pointer can only point to a certain type of variable (only to integers or only to floats, for example).
- Use a metaphor consistently; do not mix metaphors. For example, when discussing local variables, do not at one point say they are "invisible" outside their functions and at another point say that they are "nonexistent" outside their functions. Stick to one metaphor or the other.

  If you *must* use more than one metaphor, introduce transitional material and explain how and why you are switching metaphors.
- Avoid idioms and implicit metaphors wherever possible. People translate GNU documentation into many different languages. English idioms such as "this feature opens the door to the possibility of . . ." only make more work for translators whose languages do not possess the idiom.

5 English usage

Consult good books on English style. For example, a classic text is *The Elements of Style* by Strunk and White. Early editions of it are now in the public domain and are therefore free in the GNU sense.

Also consult the *GNU Coding Standards*, which discusses documentation as well as code.

- Don't mention non-free software by name unless it is unavoidable.
- Refer to GNU more and Unix less.

  Always write "GNU/Linux", never just "Linux", unless you are only referring to the Linux kernel.
- Use "illegal" only for matters of the law and government. For violations of the rules of C or other languages, use "invalid".
- Always address the reader as "you". Example: "If you want to display the diagram, press the ⟨RET⟩ key."
- Use "must", "should", "may", and "can" appropriately. Do not conflate them when discussing actions you must, should, may, or can perform while using software.
- Examples are not "given" but "shown". Only useful stand-alone programs qualify as gifts.
- "Kinds of" and "types of" are followed by a singular noun. For example:

  Bad: "kinds of computers", "types of variables"

  Good: "kinds of computer", "types of variable"
- There should be no text between "as follows" and what is said to follow.
- Be careful to separate English from C code (or the code of whatever computer language you are using). For example, in the first example below, the English word "or" might be confused by the reader with the Boolean operator OR.

  Bad: '`@dfn{logical operator} (or operator on Boolean values)`'

  Good: '`@dfn{logical operator} (an operator on Boolean values)`'
- Failure to process negatives is a common problem in reading. Phrase your text so that a reader is not likely to miss an important "not". Do not repeat the negative information in a manner that could make it appear positive.
- Distinguish computer science terms and jargon from the language of the reader's everyday experience. For example, you may need to tell the reader that the Boolean value `true` is true and only true, while in real life "true" might mean "only partly true", as in "that's a true story, although parts are exaggerated".

- Most people except LISP programmers dislike parentheses. Use as few as possible. If you can, avoid using parentheses in tables.

  Bad: unary plus (example: `+5`)

  Good: unary plus, example: `+5`

- Use language precisely. For example, when discussing C, a "declaration" is not the same as a "definition". Many distinct terms sound alike and are used in similar ways, but that is no excuse for *you* to conflate them, or to fail to distinguish them for your readers. Moreover, don't simply say that two terms are different, but explain their differences.

  Similarly, distinguish one use of a jargon word from another. For example: "value of a variable" vs. "passing by value".

6 Texinfo usage

Please read the Texinfo manual through; it will do you good.

- Use '`@code`', '`@samp`', '`@file`', etc. correctly; for clarification, see the *Texinfo Manual*.
- Use '`@xref`' properly; never use it in mid-sentence.
- To emphasize, use '`@emph`' or '`@strong`', not all-caps.
- For meta-syntactic variables, use '`@var`', not angle brackets.
- End every sentence with two spaces so Emacs can see where sentences begin and end. (See the "Sentences" section in *The GNU Emacs Manual*, which describes convenient sentence-related commands.)
- Use '`@group`' to hold together examples that should stay all on one page.

  Note that the `@group` command does not currently work with the `@table` command. Instead, use the `@need` command with `@table`. Similarly, use the `@need` command before a plain text paragraph that introduces an example or list. (See the *Texinfo Manual*.)
- Never use typesetting commands for markup! Always use logical markup instead. You gain nothing in Info by putting "Important:" in boldface. Unless your reader uses an unusual stylesheet, you will not help Emacspeak, either. Replace '`@b{Important:}`' with '`@strong{Important:}`' so formatting software can figure out how to handle the markup appropriately. The use of typesetting markup is the bane of the HTML world; logical markup works better, and one reason that XML was invented.

  The only legitimate use of a typesetting command in GNU documentation is to cause plain, explanatory text in a table or example to be in a Roman font. Use the `@r` command to do this.

7 Format As You Write

Format your text as you write. This eases your final cleanup.

Then, at the end of your project, again check how your text will look in Info, on a Web page, and typeset for printing. Listen to it as well; or, at at the very least, consider how it sounds when read out loud.

- As you write, make sure your file will compile as an Info document. In GNU Emacs, you can do the following:
  1. Run `C-c C-m C-b` (`makeinfo-buffer`), then
  2. run `C-u C-c C-u C-a` (which is `texinfo-all-menus-update`, with a prefix argument); then
  3. fix the remaining errors, then
  4. repeat this sequence until there are no more errors.
- Run the spell-checker!
- When polishing the text, make sure your page layout is attractive; for example, make sure you don't use too much whitespace. You can group chunks of text together with the `@group` and `@need` commands.
- In tables and code examples, line up columns neatly:
  Bad:

  ```
  a: 1
  b :  2
  c:3
  ```

  Good:

  ```
  a : 1
  b : 2
  c : 3
  ```

- When you email your manuscript to your editor, consider compressing it with `gzip` and sending it as a uuencoded or base-64 encoded attachment. This will prevent it from being mangled in email. Some email programs transform a '`From`' at the start of a line to '`>From`'. Gzipped and encoded attachments are not vulnerable to this sort of corruption. (Short documents without a '`From`' at the start of a line do not need to be compressed and encoded.)
- As I said before, look at your text in all three major output formats: in Info, on a Web page, and typeset for printing. In addition, listen to it; or else consider how it sounds when read out loud.