# Tip 68: Write literate code

Ben Klemens

15 February 2012

**level**: you write code that other people read
**purpose**: explain what's going on in great detail

WEB is by Donald Knuth, the author of the TEX document formatting system. He as an individual got math typesetting right some time in the 1980s; Microsoft still doesn't do it right here in 2012. TEX, in an effort to prevent obsolescence, places ink on the page at a resolution finer than visible light. It is often held up as a paragon of an extremely complicated system done very right.

Dr Knuth typically explains the hight quality of TEX by discussing how it was written: using WEB, a system that intersperses English expository text with computer code (in this case, PASCAL code). Here in the present day, PASCAL has zero percent buy-in, so the code will be in C, and now that TEX works to produce beautiful documentation, we might as well use it as the markup language for the expository side. Thus, CWEB[1].

Literate programming as described here is much more involved than the sort of documentation that Doxygen produces. Doxygen will describe the interface to every function; CWEB is for detailing how the function itself operates, procedural chunk by procedural chunk.

This isn't just about presenting documentation, either. The claim is that if you are explaining what you are doing as you are doing it, you are more likely to write higher quality code. Scroll down to the end of this post and have a look at the sample program from last time WEBified. The odds are good that wherever your eyes landed, it's on plain English text, not code.

As for the output, it's easy to find textbooks[2] that use CWEB to organize and even present the content. If somebody else is going to study your code (for some of you this might be a coworker or a review team), then CWEB might make a lot of sense.

Usage:

- Custom is to save CWEB files with a `.w` extension. Let's call the program below `headlines.w`.
- Run `cweave headlines.w` to produce a `.tex` file; then run `pdftex headlines.tex` to produce a PDF.

[1] `http://www-cs-staff.stanford.edu/~uno/cweb.html`
[2] `http://my.safaribooksonline.com/book/programming/c/9780321562807`

- Run `ctangle headlines.w` to produce a `.c` file. GNU `make` recognizes this, so `make headlines` will run `ctangle` for you.

The tangle step removes comments, which means that CWEB and Doxygen are incompatible. Perhaps you could produce a header file with a header for each public function and struct for Doxygenization, and use CWEB for your main code set.

Tips for your reading pleasure:

- Every special code for CWEB has an `@` followed by a single character. Be careful to write `@<titles@>` and not `@<incorrect titles>@`. [This might be why TEX eschews `@`s.]
- Every segment has a comment, then code. It's OK to have a blank comment, but that rhythm has to be there or else all sorts of errors turn up.
- Start a text section with a `@ ` (i.e., a space after the `@`). Then expound, using TEX formatting.
- Start an unnamed chunk of code with `@c`.
- Start a named block of code with a title followed by an equals sign (because this is a definition): `@<an operation@>=`.
- In later code, that block will get inserted verbatim wherever you use the title. That is, each chunk name is effectively a macro that expands to the chunk of code you specified (but without all the weirdness of C preprocessor macros).
- Sections (like the XML section and cURL sections below) start with `@*`, and have a title ending in a period.

That should be enough for you to read the below, and get started writing your own stuff in CWEB.

```
@ The front page of the Grey Lady's web site is as gaudy as can be, including se
  headlines and sections trying to get your attention, various formatting scheme
  and even photographs--in {\it color}.

 This program reads in the NYT headlines RSS feed, and writes a simple list in p
 HTML. You can then click through to the headline that modestly grabs your atten

@ Compiling the program

Save the following code to |makefile|.

Notice that cURL has a program, |curl-config|, that behaves like |pkg-config|,
but is cURL-specific. I do not know why they did this.

| CFLAGS =-g -Wall -std=gnu99 -O3  `curl-config --cflags`  |
```

```
| LDLIBS=‘curl-config --libs ‘ -lxml2 |

Having saved your makefile, if you name this here code file to |headlines.c|, th
use |make headlines| to compile.

@* Prep for the program.

@c
#include <stdio.h>
#include <curl/curl.h>
#include <libxml2/libxml/xpath.h>

@ Check an assertion; print a message to stderr and exit if the assertion fails.
CWEB has a @@d for |define|s, but it doesn’t recognize variadic macros.

@c
#define stopifnot(assertion, ...) if (!(assertion)){fprintf(stderr, __VA_ARGS__)

@ Here are some global variables
@c
char *rss_url = "http://feeds.nytimes.com/nyt/rss/HomePage";
char *rssfile = "nytimes_feeds.rss";
char *outfile = "now.html";

@* Dealing with XML.

@ Print a list of headlines to the outfile, which is overwritten without apology

  |urls|: The list of urls. This should have been tested for non-NULLness

  |titles|: The list of titles, also pre-tested to be non-NULL.

  If the |urls| list or the |titles| list is |NULL|, this will crash.

@c
void print_to_html(xmlXPathObjectPtr urls, xmlXPathObjectPtr titles){
    FILE *f = fopen(outfile, "w");
    for (int i=0; i< titles->nodesetval->nodeNr; i++)
        @<print one URL/title@>
    fclose(f);
}


@ Given a matched list of URL nodes and title nodes, print their text
  to the output file.
```

```
@<print one URL/title@>=
    fprintf(f, "<a href=\"%s\">%s</a><br>\n"
             , xmlNodeGetContent(urls->nodesetval->nodeTab[i])
             , xmlNodeGetContent(titles->nodesetval->nodeTab[i]));



@
Parse an RSS feed on the hard drive. This will parse the XML, then find all node
matching the XPath for the title elements and all nodes matching the XPath for t
Then, it will write those to the outfile.

  The |infile| is the RSS file to read.

@c
void parse(char *infile){
    @<open an XML file@>@;
    @<get a context pointer@>@;
    @<get elements matching two paths@>@;
    print_to_html(urls, titles);
    @<cleanup XML@>@;
}

@ The next few chunks are largely self-explanatory bits of the XML-reading proce
        their comment sections are blank).

@<open an XML file@>=
    xmlDocPtr doc = xmlParseFile(infile);
    stopifnot(doc, "Error: unable to parse file \"%s\"\n", infile);

@
@<get a context pointer@>=
    xmlXPathContextPtr context = xmlXPathNewContext(doc);
    stopifnot(context, "Error: unable to create new XPath context\n");

@
@<get elements matching two paths@>=
const xmlChar *titlepath= (xmlChar*)"//item/title";
const xmlChar *linkpath= (xmlChar*)"//item/link";
    xmlXPathObjectPtr titles = xmlXPathEvalExpression(titlepath, context);
    xmlXPathObjectPtr urls = xmlXPathEvalExpression(linkpath, context);
    stopifnot(titles && urls, "either the Xpath '//item/title' or '//item/link'

@
@<cleanup XML@>=
    xmlXPathFreeObject(titles);
```

```
    xmlXPathFreeObject(urls);
    xmlXPathFreeContext(context);
    xmlFreeDoc(doc);
```

```
@* Getting the RSS feed.

@
Use cURL's easy interface to download the current RSS feed.

|url|: The URL of the NY Times RSS feed. It may be that any of the ones listed a
            @.http://www.nytimes.com/services/xml/rss/nyt/@> will work.

|outfile|: The headline file to write to your hard drive. I'll first save the
        RSS feed to this location, then overwrite it with the short list of link

return: 1==OK, 0==failure.

@<curl's function@>=
int get_rss(char *url, char *outfile){
        @<open the outfile@>@;
        @<set cURL options@>@;
        @<cURL go!@>@;
        @<cURL cleanup@>@;
}

@ Open the file to write to as usual.
@<open the outfile@>=
    FILE *feedfile = fopen(outfile, "w");
    if (!feedfile) @<send error@>;

@ cURL provides an 'easy' interface with fewer options for when you just
want to get a file via some protocol. It basically consists of setting up a few
pieces of information about how to get the data and where to put it.
@<set cURL options@>=
    CURL *curl = curl_easy_init();
    if(!curl) @<send error@>;
    curl_easy_setopt(curl, CURLOPT_URL, url);
    curl_easy_setopt(curl, CURLOPT_WRITEDATA, feedfile);

@ OK, everything is set up; let's call up the server!
@<cURL go!@>=
    CURLcode res = curl_easy_perform(curl);
    if (!res) @<send error@>;
```

```
@ close cURL's connection and the outfile.
@<cURL cleanup@>=
    curl_easy_cleanup(curl);
    fclose(feedfile);
    @<send OK@>;


@ This is a fragment of text used by name above.
@<send error@>=
return -1

@
@<send OK@>=
return 0

@ Main is at this point a two-step: get the RSS feed, and parse it.
We could turn these into WEB macros as well, but I saw little benefit.

@c
@<curl's function@>
int main(void) {
    stopifnot(get_rss(rss_url, rssfile), "failed to download %s to %s.\n", rss_u
    parse(rssfile);
}
```