literac

A Program That Enables Literate Commenting

Doug McKenna
Mathemaesthetics, Inc.
Boulder, Colorado

TUG — 2014

Literate Programming

- The phrase was introduced by Don Knuth 30 years ago
- Memo: "The `WEB` System of Structured Documentation", Stanford Univ., 1983
- An amalgam of Pascal and TEX, with its own added layer of markup commands
- Two post-processing tools—`TANGLE` and `WEAVE`—create separate Pascal code and TEX documentation files
- `CWEB` and tools created in 1991 by Knuth and Levi
- `web2c`, by Tom Rokicki, processes `WEB` code directly to C code
- TEX's source code is still written in `WEB`, 30 years later

Problems that WEB solved

- Formal conversion and re-arrangement of pseudo-code constructs into code
- Enriched reading experience: fonts and TEX-quality typesetting
- Code and documentation more easily kept synchronized
- Automatic upward movement of code lines from the best expositional spot to a correct compilable spot
- No macro preprocessor in Pascal language
- Simple syntax for typesetting code |snippets| inside comments
- Treats a large program as a piece of literature worth reading (at least, if one writes well)
- Automated document features (table of contents, index, etc.)

Why do most programmers not use WEB (or CWEB)?

- Markup is terse, undiscoverable, not quite free-form
- Requires—rather than gently permits—the programmer to think in several languages at once
- The WEB/TANGLE/WEAVE bootstrap keeps users away
- Fosters the use of global variables; turns locals into "globals"
- Complicates and slows down the edit-compile-test cycle for working programmers (in the zone)
- Code is typeset as mathematical notation, not as code
- Solutions should be in the computer language and IDE editors
- Search the internet for "literate programming" and "failure" for more

As a Programmer, I Want Literate Commenting with

- Sweet incremental simplicity, reasonable power, but no lock-in
- Documentation derived from source code, not vice-versa
- Documenting independent of edit-compile-test cycle
- Only a few, innocuous markup commands to remember
- Original source still readable/understandable after markup
- Code format/style inviolate (no pretty-printing; yes long lines)
- Defaults for immediate success or "good enough" solutions
- Decent higher-level error reporting and recovery
- No fighting against TEX/LATEX ignorance/confusion
- Access to LATEX's power if I need it (and know what I'm doing)
- An inviting, literate, well-typeset exposition of my program
- A LATEX file to modify further, if I want or need to

What is "literac"?

- literac is (currently) a command-line program, written in C
- Written to "codify" commenting conventions in large C library
- Comprises 6000 lines of code, 6000 of literate commenting
- Eats its own dog food to create manual and literate program
- Processes about 100,000 lines of C source code per second
- Supports multiple input files and options in one invocation
- Outputs one or more LaTeX files that can be immediately run
- Needs `fancyvrb`, `dashrule`, and other standard packages
- Does not rely on `listings` or similar packages
- Currently, only supports comments using `/*...*/` or `//...`
- Languages: C, C++, Objective-C, Go, Swift, and few others
- Handles obscure edge cases and (some) commenting idioms

Typeset Comments Don't Need No Stinkin' Delimiters

- Delimiters `//` and `/*...*/` are for the benefit of compiler
- And they are for the benefit of source code author (initially)
- But they are (usually) unnecessary for a reader
- Delimiters are redundant in editors that do syntax coloring
- Delimiters in source code are thus *syntactic noise*
- They interfere with vertical eye scanning of left edge of code
- Typesetting is about visual hints on behalf of meaning
- So ... literac gets rid of all delimiters, unless doing so would introduce ambiguity
- Comments must therefore use different type styles from code
- Code is easy: get it into a `verbatim` fixed-width code font
- literac focuses on comments much more than code

C-style Comment Taxonomy

Two classes of comment delimiter: block and gloss

- Gloss comments use `//` and the end of same line (usually)
- Block comments use `/* ... */` (possibly multiple lines)

- Also pseudo-gloss: `/* ... // ... */`
- And pseudo-block: `// ... /* ... */`

- Nested block: `/* ... /* ... */ ... */` (Swift only) (not yet supported)

Comment Taxonomy – Rest-of-Line Gloss Comments

Gloss-only (after indentation, comment on entire line)

- `//`
- `// text`
- `// text\`
  `more text`

Code then gloss on remainder of line

- `foo = bar(n);` `//`
- `bar = foo(n);` `// text`
- `foo = bar(n);` `// text\`
  `more text`

Comment Taxonomy – Single Line Block Comments

- `/**/`
- `/*  */`
- `/* text */`
- `/* text\`
  `more text */`
- `foo = bar(n);  /* text */`
- `foo = bar(n);  /* text */  /* more text */`
- `/* text */  bar = foo(n);  /* more text */`
- `foo = bar( /* text */ n);`

Comment Taxonomy – Simple Block Comments

- ```
  /*

  */
  ```
- ```
  /*
  text (delimiters are on their own lines)
  */
  ```
- ```
  /*
          indented text
  */
  ```
- ```
  /*
   * text
   * more text
   * yet more text
   * and a vertical * bar
   */
  ```

Comment Taxonomy – Quiet Block Comments

Simple block comments with delimiters far to the right:

- ```
  A line of commenting text.                                    */
  ```

- ```
  A line of commenting text.                                    /*
  Another line of commenting text.
                                                                */
  ```

Style reduces syntactic noise on left edge of source code

Comment Taxonomy – Complex Block Comments

/* or */ occurs on same line as some comment text or code

- ```
  /* text ...
  ... more text */
  ```
- ```
  foo = far(n);       /* text
                         more text */
  ```
- ```
  /* text
  more text */      foo = far(n);
  ```

literac works to regularize these into simple block comments
  ```

Comment Taxonomy – ASCII Art Block Comments

- Lines or boxes made of * to create poor man's rules
- ```
  /**************************************/
  /* And In This Section of the Program */
  /**************************************/
  ```
- ```
  /**************************************\
   * And In This Section of the Program *
  \**************************************/
  ```
  (this second example abuses line continuation on first line)
- literac erases the bars (currently doesn't replace with any rules, but might in the future) to regularize

What literac does

- Classify each input line's start as "in code" or "in comment"
- Determine whether line ends in code or comment
- Honor line continuation only for comment line ends, not code
- Divide line into two (possibly empty) areas: code or comment
- Typeset code area on left using verbatim fixed-width font
- Strip comment delimiters, unless in code area, or if ambiguity
- Execute all `literac` commands in remaining comment text
- Converts dividers to rules, and manages a table of contents
- Prevent TEX from getting confused by special characters
- Attempt to do smart-quoting in both code and comment
- Let TEX merge "similar" comment lines into paragraphs
- Comment vs. commentary styles, based on indentation or not

Special Delimiters Available Within Block Comments

When typesetting block comment lines, `literac` responds to the following patterns when they appear by themselves (after any indentation, but no other text) on any line:

- `/*` Start a block comment, delete line if not indented
- `*/` End a block comment, delete line if not indented
- `\\` Toggle one-liner mode; delete line if not indented

If line not deleted, it becomes blank and is honored as such.

- `|@` enters pure TEX line collection mode; line is deleted
- `@|` exits pure TEX line collection mode; line is deleted

Pure TEX collection mode allows the injection of 0 or more lines of arbitrary TEX or LATEX code, exactly as if in a `".tex"` file.

Super Gloss Comments

A super gloss comment tells `literac` to delete the delimiter and the rest of the line from the typeset code. There are three variants:

- `///` Delete rest of line, trim whitespace, delete line if empty
- `//@` Same, but doesn't conflict with, e.g., `Doxygen`
- `//.` Delete rest of line, trim, but leave line blank if empty

These are good for commenting out code that serves no purpose in the typeset version.

Good for issuing `literac` gloss comment commands on lines that won't be typeset as blank.

Special Commands Within Block or Gloss Comment Text

`literac` commands consist of an identifier immediately followed by a (possibly empty) brace-enclosed argument. Everything must (currently) be on one comment line. These can occur anywhere in a block or gloss comment's text.

- `emph{` *text* `}` Emphasize *text* (uses `\emph` in LaTeX file)
- `bold{` *text* `}` Put *text* into bold (uses `\bfseries`)
- `math{`*formula*`}` Typeset *formula* inside `$...$` math mode
- `Math{`*formula*`}` Same, but use a `$$...$$` math display
- `text{` *line* `}` Prevent short *line* from being a divider title
- `toc{` *title* `}` Insert table of contents labeled with *title*

Without an immediate left brace, it's just more comment text.

Verbatim Quoting in Comment Text

The usual `WEB` and LaTeX verbatim syntax using `|`:

Within any comment text, block or gloss, use a pair of `|`s to typeset a code snippet, as long as the entire quote is on one line.

- Relies on the `fancyvrb` package (except in titles)
- But ... in `literac` any single character between two `|`s will be verbatimed, no exceptions, no escapes
- `|||` is the verbatim quote of a single `|`
- `|\|` is the verbatim quote of a single `\`
- `||` is the empty verbatim quote (if not followed by another `|`)
- `|@|` is the verbatim quote of a single `@`
- `|@` *code* `@|` injects pure TeX *code* directly into comment's vein

Special Gloss Comment Commands

Within any gloss, pseudo-gloss, or super-gloss comment, `literac` processes and then deletes 0 or more gloss comment commands:

- `verbatim{` *name1 name2 . . .* `}`
- `define{[ ]`*name*`[ ]}` *definition*
- `needlines{` *n* `}`
- `skip{` *n* `}` or `skip{}`
- `save{` *line group name* `}`
- `done{}`
- `insert{` *line group name* `}`
- `pushoption{` *option list* `}`
- `popoption{}`

Predefining Names to Be Auto-Verbatimed

`literac` currently doesn't parse code looking for variable, type, or macro names. But you can declare any set of alphanumeric identifiers to be automatically typeset `|verbatim|`, using

```
/// verbatim{ name1 name2 ... }
```

Additionally, if a name begins with a capital letter, its (English) plural form (ending in "s" or "es"), will be typset in the singular form, with the pluralization in non-verbatim style, e.g.,

```
/// verbatim{Entry}
// Of these Entrys, only the last counts
```

will typeset as

Of these `Entrys`*, only the last counts*

Defining a Simple Macro

When in a comment `literac` recognizes a name that has been earlier defined as a `literac` macro, the macro's definition text is substituted in its place. Names are defined with:

`/// define{[ ]`*name*`[ ]}` *definition*

The *definition* is whatever text remains on the line after the }. Whitespace on either side of *name* inside braces matches one space character before or after *name* in a comment's text.

`literac` macros currently don't support argument lists.

Example: `/// define{TheFile} |@\texttt{"bingo.c"}@|`

Both macro and auto-verbatim names can be preceded by an empty verbatim quote `||` to prevent expansion or style changing.

Ensuring Page Has *n* More Lines Available

The `needlines{}` gloss comment command can be used to force the next few lines onto the start of the next page, if there's not enough room at the bottom of whatever current page they are otherwise ending up on. For example,

```
/// needlines{7}
```

ensures that the next 7 comment lines will remain unbroken by a page break.

Useful with one-liner mode, and to push subroutine starts to the next page or keep lines for short pedagogical examples together.

Preventing Compilable Code from Being Typeset

The `skip{}` gloss comment command declares the start of a sequence of code or comment lines (or both) that should be supressed from being typeset. The command has two forms.

```
// skip{3}
```

prevents 3 lines from being typeset, including the line the `skip` command is on.

```
// skip{}
```

suppresses all lines until a `done{}` command is executed.

Good for suppressing C prototypes (forward reference noise).

Presenting Code in Any Language Verbatim, in a C File

A `skip{1}` command, in conjunction with C preprocessor `#if 0` and `#endif`, lets your C source file present code, typeset verbatim, in any language whatsoever. For example,

```
/*
#if 0           // skip{1}

    Some \LaTeX\ code\par    % with TeX comment

    reverse = sort(names, $0 > $1)     // Swift

#endif          // skip{1}

*/
```

can be used to show many lines of verbatim code (here, in the TEX and Swift languages) between two literate block comments in your C program, but typeset without all the syntactic noise.

Postponing Code or Comments until Later

`WEB` migrates source lines upward for earlier compilation.
`literac` can migrate source lines downward for later typesetting.
Both strategies are in the service of top-down explanations.

A `save{` *line group name* `}` command lets you collect lines of code or comment under a *line group name*, to be removed and then later reinserted at an appropriate place in your exposition of the program. For example,

```
typedef struct foo {     // save{FooBarStuff}
    int flag;
    struct foo *next;
    }                            // done{}
```

will suppress the `struct` declaration's four lines from being typeset where it is found in the source file, but saves the four lines for later insertion during typesetting by invoking the name `FooBarStuff`.

Inserting Postponed Lines of Code or Comment

The `insert{` *line group name* `}` command lets you insert lines of code or comment that have been previously saved under a *line group name*. For example,

```
/// insert{FooBarStuff}
```

will place the lines previously saved under the name `FooBarStuff` on the input to be read again and typeset at a new position.

Because commands are deleted from comments after execution, re-examining a line won't re-execute any commands.

(If visible, line numbers will be marked to indicate that they are typeset out of order from that in the original source code.)

Specifying Titled Chapters, Sections, Subsections, etc.

Within a block comment, you can start any line with a sequence of characters that `literac` recognizes as a divider.

If a line inside a block comment starts with

- `####` ... it's a part divider (with title on next line).
- `====` ... it's a chapter divider (with title on next line).
- `++++` ... it's a section divider (with optional title).
- `----` ... it's a subsection divider (with optional title).
- `- - -` .. it's a subsubsection divider (with optional title).
- `.` (a lone period), creates a blank line "divider".

Each divider (except the blank line) has a searchable, "tagged" variant to enter any title into a table of contents, inserted earlier with a `toc{` *title* `}` command.

Integrating a README or Manual into Implementation

Using the `-readme` command line option, `literac` will recognize the following super-gloss comment

```
////\\\\////\\\\ ...
```

as a signal to stop typesetting and ignore the rest of the file.

For example, `literac.c` uses one of these comments to automatically convert itself to its own LaTeX-typesettable user manual, ignoring the nuts and bolts of the implementation.

Command-line Options

- -help or -h
- -readme
- -linenumbers -nolinenumbers
- -strip -nostrip -stripblock -stripgloss
- -nofile
- -silent -verbose
- -clean
- -tab <n> -intab <n> -outtab <n>
- -maxline <n>
- -noplural -plural

Managing `literac` Command-line Options in Comments

Use the `pushoption{...}` and `popoption{}` commands to save, change, override, and restore the command line options that `literac` has been invoked with.

```
/// pushoption{linenumbers=off} -- suppress them
/// popoption{} -- restore previous settings
```

Currently, only turning line numbers on or off is supported.

To-Do List (as of July 2014)

- Customized LaTeX preamble
- A formal `literac`-support package for LaTeX
- More command-line option push/pop support
- Concatenating input files to one output LaTeX file
- Shared macro dictionary across multiple files
- More leveraging off of `fancyhdr` and `hyperref`
- Automatic verbatiming of identifier or type names
- Conversion of '*' patterns to actual rules or frames
- Indexing
- Other language's comment delimiters
- ...

Testing `literac` on Its Own Source Code

- `"literac.c"` is commented using its own syntax
- Examples in manual are authentic