Website Maker 3/web

Purpose

To accompany a release with a mini-website.

3/web.§1-6 Styling with CSS; §7-9 Making an HTML page from a template; §10 Rendering the source text as HTML pages; §11-19 Pass 1: scanning the source for tables and headings; §20-55 Pass 2: writing the source text pages

---

Definitions

¶1. Making a website is not especially tricky. The difficult part is typesetting the source text into it, if that's been requested. We will need to do that by scanning the source text for typographically significant structures:

```
define ABBREVIATED_HEADING_LENGTH 1000

typedef struct table {
    int table_line_start;                    line number in the source where the table heading appears
    int table_line_end;                      line number of the blank line which marks the end of the table body
    MEMORY_MANAGEMENT
} table;

typedef struct heading {
    int heading_line;                        line number in the source at which the heading appears
    int heading_level;                       a low number makes this a more significant heading than a high number
    int heading_has_content;                 is there anything other than white space before the next heading?
    struct segment *heading_to_segment;      which segment contains the heading
    char heading_text[ABBREVIATED_HEADING_LENGTH + 1];    truncated if necessary for the contents
    MEMORY_MANAGEMENT
} heading;
```

The structure table is private to this section.

The structure heading is private to this section.

¶2. Segments are used to divide the source text into pieces of what we hope will be a manageable size.

It is not true that the source text is partitioned exactly by segments. The topmost segment begins at the first heading in the source text. So there will usually be at least a few prefatory lines before this point – perhaps the title, some extension inclusions, and so on – and it's even possible, if there are no headings at all, for there to be no segments so that the entire source text is "prefatory". If we have three segments, then, we will split the source text into four HTML files:

`source0.html` – "Page 1 of 4", the preface and then contents
`source1.html` – "Page 2 of 4", first segment (with allocation ID 0)
`source2.html` – "Page 3 of 4", second segment (with allocation ID 1)
`source3.html` – "Page 4 of 4", third segment (with allocation ID 2)

Note that the prefatory lines contain no headings, that every heading belongs to a unique segment (hence the `heading_to_segment` field above) and that the top line of every segment is always a heading. A single segment can contain multiple headings, because we run on a heading if it contains no content except white space: this is so that, e.g.,

Part I - Up the Amazon

Section I.1 - The lower delta

Rickety Jetty is a room. [...]

would be combined into a single segment, rather than a pointlessly short segment just containing the "Part I" heading followed by a second segment opening with "Section I.1".

```
typedef struct segment {
    int begins_at;                                          line number on which the segment begins
    int ends_at;      line number of the last line of the segment, or MAX_SOURCE_TEXT_LINES if it runs to the end
    int documentation;                                      is this in the documentation of an extension?
    struct text_file_position start_position_in_file;                                  within the source text
    struct heading *most_recent_heading;                                  or NULL if there hasn't been one
    struct table *most_recent_table;                                      or NULL if there hasn't been one
    char segment_url[MAX_FILENAME_LENGTH];
    char *link_home;
    char *link_contents;
    char *link_previous;
    char *link_next;
    int page_number;
    MEMORY_MANAGEMENT
} segment;
```

The structure segment is private to this section.

---

§**1. Styling with CSS.** We try to give the template files as much freedom as possible to define whatever CSS styles they need, but the template can't see inside the text of variables, so `cblorb` itself has to choose CSS styles for anything interesting that is displayed there. We use the following style names, which a CSS file is required to define:

`columnhead` – the heading of a column in a Table in I7 source text
`comment` – comments in I7 source text
`filetype` – the "(pdf, 150KB)" text annotating links
`heading` – heading or top line of a Table in I7 source text
`i6code` – verbatim I6 code in I7 source text
`notecue` – footnote cues which annotate I7 source text
`notesheading` – the little "Notes" subheading above the footnotes to source text
`notetext` – texts of footnotes which annotate I7 source text
`quote` – double-quoted text in I7 source text
`substitution` – text substitution inside double-quoted text in I7 source text

In addition it must provide paragraph classes `indent0` to `indent9` for code which begins at tab positions 0 to 9 (see below). Although "Standard.css" contains other names of classes, these are only needed because "Standard.html" or "Standard-Source.html" say so: `cblorb` does not mandate them.

§**2.** In case CSS is not available, we use old-fashioned HTML alternatives:

```
void open_style(FILE *write_to, char *new) {
    if (new == NULL) return;
    if (use_css_code_styles) {
        fprintf(write_to, "<span class=\"%s\">", new);
    } else {
        if (strcmp(new, "columnhead") == 0) fprintf(write_to, "<u>");
        if (strcmp(new, "comment") == 0) fprintf(write_to, "<font color=#404040>");
        if (strcmp(new, "filetype") == 0) fprintf(write_to, "<small>");
        if (strcmp(new, "heading") == 0) fprintf(write_to, "<b>");
        if (strcmp(new, "i6code") == 0) fprintf(write_to, "<font color=#909090>");
        if (strcmp(new, "notecue") == 0) fprintf(write_to, "<font color=#404040><sup>");
        if (strcmp(new, "notesheading") == 0) fprintf(write_to, "<i>");
        if (strcmp(new, "notetext") == 0) fprintf(write_to, "<font color=#404040>");
        if (strcmp(new, "quote") == 0) fprintf(write_to, "<font color=#000080>");
        if (strcmp(new, "substitution") == 0) fprintf(write_to, "<font color=#000080>");
    }
}
void close_style(FILE *write_to, char *old) {
    if (old == NULL) return;
    if (use_css_code_styles) {
        fprintf(write_to, "</span>");
    } else {
        if (strcmp(old, "columnhead") == 0) fprintf(write_to, "</u>");
        if (strcmp(old, "comment") == 0) fprintf(write_to, "</font>");
        if (strcmp(old, "filetype") == 0) fprintf(write_to, "</small>");
        if (strcmp(old, "heading") == 0) fprintf(write_to, "</b>");
        if (strcmp(old, "i6code") == 0) fprintf(write_to, "</font>");
        if (strcmp(old, "notecue") == 0) fprintf(write_to, "</sup></font>");
        if (strcmp(old, "notesheading") == 0) fprintf(write_to, "</i>");
        if (strcmp(old, "notetext") == 0) fprintf(write_to, "</font>");
        if (strcmp(old, "quote") == 0) fprintf(write_to, "</font>");
        if (strcmp(old, "substitution") == 0) fprintf(write_to, "</font>");
    }
}
```

The function open_style is called from 3/links.
The function close_style is called from 3/links.

§**3.** In what follows, we will need to have a current typographic style for text, and may need to change it at any point inside the paragraph. We represent the current style by the global variable `current_style`, which is either `NULL` (for ordinary text) or the name of one of the styles above.

```
char *current_style = NULL;
void change_style(FILE *write_to, char *new) {
    if (current_style) close_style(write_to, current_style);
    open_style(write_to, new);
    current_style = new;
}
```

The function change_style is.

§**4.** We also use CSS to manage code indentation, when it's available, since this can handle hanging indentation much better.

The block of source text displayed on a web page should be framed within:

```
void open_code(FILE *write_to) {
    if (use_css_code_styles == FALSE) {
        fprintf(write_to, "<p>");
    }
}
void close_code(FILE *write_to) {
    if (use_css_code_styles == FALSE) {
        fprintf(write_to, "</p>");
    }
}
```

The function open_code is.
The function close_code is.

§**5.** Each individual paragraph of the source text (which looks like a line to us) should then be framed within:

```
void open_code_paragraph(FILE *write_to, int indentation) {
    if (use_css_code_styles) {
        char *classname = "";
        switch (indentation) {
            case 0: classname = "indent0"; break;
            case 1: classname = "indent1"; break;
            case 2: classname = "indent2"; break;
            case 3: classname = "indent3"; break;
            case 4: classname = "indent4"; break;
            case 5: classname = "indent5"; break;
            case 6: classname = "indent6"; break;
            case 7: classname = "indent7"; break;
            case 8: classname = "indent8"; break;
            default: classname = "indent9"; break;
        }
        fprintf(write_to, "<p class=\"%s\">", classname);
    } else {
        int i;
        for (i=0; i<indentation; i++) fprintf(write_to, "    ");
    }
}
void close_code_paragraph(FILE *write_to) {
    if (use_css_code_styles) {
        fprintf(write_to, "</p>");
    } else {
        fprintf(write_to, "<br/>");
    }
}
```

The function open_code_paragraph is.
The function close_code_paragraph is.

§**6.** In the age of CSS, old-fashioned elements like `halign` for individual table cells are deprecated, so:

```
void open_table_cell(FILE *write_to) {
    if (use_css_code_styles) {
        fprintf(write_to, "<td>");
    } else {
        fprintf(write_to, "<td halign=\"left\" valign=\"top\">");
    }
}
void close_table_cell(FILE *write_to) {
    if (use_css_code_styles) {
        fprintf(write_to, "</td>");
    } else {
        fprintf(write_to, "  </td>");
    }
}
```

The function open_table_cell is.
The function close_table_cell is.

§7. Making an HTML page from a template.

```
FILE *COPYTO = NULL;
void web_copy(char *from, char *to) {
    if ((from == NULL) || (to == NULL) || (strcmp(from, to) == 0))
        fatal("files confused in website maker");
    HTML_pages_created++;
    COPYTO = fopen(to, "w");
    if (COPYTO == NULL) { error_1("unable to open file to be written for web site", to); return; }
    file_read(from, "can't open template file", FALSE, copy_html_line, 0);
    fclose(COPYTO);
}
```

The function web_copy is called from 1/main and 3/rel.

§**8.** Each line in turn comes here, then:

```
void copy_html_line(char *line, text_file_position *tfp) {
    int i;
    for (i=0; line[i]; i++) {
        ⟨Detect square-bracketed names of Web variables and expand them 9⟩;
        fprintf(COPYTO, "%c", line[i]);
    }
    fprintf(COPYTO, "\n");
}
```

The function copy_html_line is.

§**9.**

⟨Detect square-bracketed names of Web variables and expand them 9⟩ ≡

```
if (line[i] == '[') {
    int j;
    for (j=i+1; (line[j] && line[j]!=']'); j++) ;
    if (line[j] == ']') {
        line[j] = 0; copy_placeholder_to(line+i+1, COPYTO); line[j] = ']';
        i = j;
        continue;
    }
}
```

This code is used in §8.

§**10. Rendering the source text as HTML pages.** This is a fiddly operation, which requires us to parse the source text and then typeset it appealingly in a whole suite of HTML pages. This necessarily involves loops, but our main aim is to complete the process in $O(N)$ running time, where N is the number of lines in the source text. (Note that the number of HTML files to be written will also be $O(N)$.)

This is done in two passes. On pass 1, we scan the source text for tables and headings, and divide the whole into "segments", each of which is typeset as a single HTML page: segments do not quite correspond to headings, as we shall see. But we write nothing. On pass 2, we actually write these HTML pages.

```
char source_text[MAX_FILENAME_LENGTH];
void web_copy_source(char *template, char *website_pathname) {
    strcpy(source_text, read_placeholder("SOURCELOCATION"));
    scan_source_text();
    write_source_text_pages(template, website_pathname);
}
```

The function web_copy_source is called from 3/rel.

§**11. Pass 1: scanning the source for tables and headings.** During this scan, we will maintain the following variables:

```
int within_a_table;                          are we inside a Table declaration in the source text?
int scan_quoted_matter;                      are we inside double-quoted matter in the source text?
int scan_comment_nesting;                    level of nesting of comments in source text: 0 means "not in a comment"
text_file_position *latest_line_position;    ftell-reported byte offset of the start of the current line in the source
table *current_table;                        the Table which started most recently, or NULL if none has
heading *current_heading;                    the heading seen most recently, or NULL if none has been
segment *current_segment;                    the segment which started most recently, or NULL if none has
int position_of_documentation_bar;           line count of the ---- Documentation ---- line, if there is one
```

§**12.** Pass 1 has running time $O(N)$ since it calls `scan_source_line` exactly once for each line in the source, and `scan_source_line` looks only at a single line and at the current table, heading and segment.

```
void scan_source_text(void) {
    within_a_table = FALSE;
    scan_comment_nesting = 0;
    scan_quoted_matter = FALSE;
    latest_line_position = NULL;
    current_table = NULL;
    current_heading = NULL;
    current_segment = NULL;
    position_of_documentation_bar = MAX_SOURCE_TEXT_LINES;
    file_read(source_text, "can't open source text of project", TRUE, scan_source_line, NULL);
    ⟨Adjust heading levels downwards as far as we can without losing relative hierarchy 13⟩;
}
```

The function scan_source_text is.

§**13.** Suppose our source contains only headings at levels 3 and 4: we can reduce these to levels 0 and 1 without disturbing their relative importance, and that makes it easier for us to typeset them in a sensible way – there's no point making any typographic allowance for three sizes of headings greater than are found anywhere in the source text.

⟨Adjust heading levels downwards as far as we can without losing relative hierarchy 13⟩ ≡

```
    int minhl = 10;
    heading *h;
    LOOP_OVER(h, heading)
        if (h->heading_level < DOC_LEVEL)
            if (h->heading_level < minhl)
                minhl = h->heading_level;
    LOOP_OVER(h, heading)
        if (h->heading_level < DOC_LEVEL)
            h->heading_level -= minhl;
```

This code is used in §12.

§**14.** Here we scan each single line. (Lines to us may look like whole paragraphs to the Inform user; we're dealing with gaps between explicit line break characters.)

```
void scan_source_line(char *line, text_file_position *tfp) {
    int lc = tfp_get_line_count(tfp), lv = DULL_LEVEL;
    latest_line_position = tfp;
    if (scan_quoted_matter == FALSE)
        ⟨Look at the first word on the line to find the level of our interest 15⟩;
    if ((scan_comment_nesting > 0) && (lv != EMPTY_LEVEL)) lv = DULL_LEVEL;
    ⟨Correct the comment nesting level ready for next time 16⟩;
    if ((lv == DULL_LEVEL) && (current_heading)) current_heading->heading_has_content = TRUE;
    if ((lv == EMPTY_LEVEL) && (within_a_table)) ⟨End a table here and return 18⟩;
    if (lv == TABLE_LEVEL) ⟨Start a new table here and return 17⟩;
    if ((lv == EMPTY_LEVEL) || (lv == DULL_LEVEL)) return;
    if (lv == DOC_LEVEL) position_of_documentation_bar = lc;
    ⟨Place a new heading here 19⟩;
}
```

The function scan_source_line is.

§**15.** Looking at the first word, if any, tells whether we are a heading, or the start of a table, or an empty line, or none of these (in which case a line is perhaps unfairly called "dull"). We set `lv` accordingly.

```
define EMPTY_LEVEL -1
define DULL_LEVEL 0
define TABLE_LEVEL 1000
define DOC_LEVEL 1001
define EXAMPLE_LEVEL 1002
define DOC_CHAPTER_LEVEL 1003
define DOC_SECTION_LEVEL 1004
```

⟨Look at the first word on the line to find the level of our interest 15⟩ ≡

```
    char fword[32];
    extract_word(fword, line, 32, 1);
    if (fword[0] == 0) lv = EMPTY_LEVEL;
    if (strcmp(fword, "table") == 0) lv = TABLE_LEVEL;
    if (lc > position_of_documentation_bar) {
        if (strcmp(fword, "chapter:") == 0) lv = DOC_CHAPTER_LEVEL;
        if (strcmp(fword, "section:") == 0) lv = DOC_SECTION_LEVEL;
        if (strcmp(fword, "example:") == 0) lv = EXAMPLE_LEVEL;
    } else {
        if (strcmp(fword, "volume") == 0) lv = 1;
        if (strcmp(fword, "book") == 0) lv = 2;
        if (strcmp(fword, "part") == 0) lv = 3;
        if (strcmp(fword, "chapter") == 0) lv = 4;
        if (strcmp(fword, "section") == 0) lv = 5;
        if (strcmp(fword, "----") == 0) {
            extract_word(fword, line, 32, 2);
            if (strcmp(fword, "documentation") == 0) {
                extract_word(fword, line, 32, 3);
                if (strcmp(fword, "----") == 0) lv = DOC_LEVEL;
            }
        }
    }
```

This code is used in §14.

§**16.**

⟨Correct the comment nesting level ready for next time 16⟩ ≡

```
    int i;
    for (i=0; line[i]; i++) {
        if (line[i] == '[') scan_comment_nesting++;
        if (line[i] == ']') scan_comment_nesting--;
        if ((scan_comment_nesting == 0) && (line[i] == '\"'))
            scan_quoted_matter = (scan_quoted_matter)?FALSE:TRUE;
    }
```

This code is used in §14.

§17.

⟨Start a new table here and return 17⟩ ≡

```
current_table = CREATE(table);
current_table->table_line_start = lc;
current_table->table_line_end = MAX_SOURCE_TEXT_LINES;
within_a_table = TRUE;
return;
```

This code is used in §14.

§18.

⟨End a table here and return 18⟩ ≡

```
current_table->table_line_end = lc;
within_a_table = FALSE;
return;
```

This code is used in §14.

§19.

⟨Place a new heading here 19⟩ ≡

```
heading *new_h = CREATE(heading);
strncpy(new_h->heading_text, line, ABBREVIATED_HEADING_LENGTH);
(new_h->heading_text)[ABBREVIATED_HEADING_LENGTH] = 0;
new_h->heading_level = lv;
new_h->heading_line = lc;
new_h->heading_has_content = FALSE;
if ((current_heading == NULL) || (current_heading->heading_has_content) ||
    (lv == DOC_LEVEL)) {
    if (current_segment) current_segment->ends_at = lc - 1;
    current_segment = CREATE(segment);
    current_segment->begins_at = lc;
    current_segment->ends_at = MAX_SOURCE_TEXT_LINES;
    current_segment->start_position_in_file = *latest_line_position;
    current_segment->most_recent_heading = current_heading;
    current_segment->most_recent_table = current_table;
    current_segment->documentation = FALSE;
    if (lc >= position_of_documentation_bar) current_segment->documentation = TRUE;
}
new_h->heading_to_segment = current_segment;
current_heading = new_h;
```

This code is used in §14.

§20. Pass 2: writing the source text pages. Though there is no obvious way that the following routine passes control to the routines below it, in fact it does: `web_copy` works on the template and finds reserved variables such as "[SOURCE]"; expanding those then calls the routines below.

```
segment *segment_being_written = NULL;
int no_doc_files = 0, no_src_files = 0;

void write_source_text_pages(char *template, char *website_pathname) {
    char contents_page[MAX_FILENAME_LENGTH];
    sprintf(contents_page, "%s%c%s.html", website_pathname, SEP_CHAR,
        read_placeholder("SOURCEPREFIX"));
    char *contents_leafname = get_filename_leafname(contents_page);
    ⟨Devise URLs for the segments 21⟩;
    ⟨Work out how the segments link together 22⟩;
    ⟨Generate the prefatory page, which isn't a segment 23⟩;
    ⟨Generate the segment pages 24⟩;
}
```

The function write_source_text_pages is.

§21. Calling these URLs is a bit grand, since they are only leafnames. The source segments have pages `source_0.html` and so on up; the documentation pages `doc_0.html` and so on up.

⟨Devise URLs for the segments 21⟩ ≡

```
    segment *seg;
    LOOP_OVER(seg, segment) {
        segment_being_written = seg;
        if (seg->documentation) {
            sprintf(seg->segment_url, "doc_%d.html", no_doc_files++);
            seg->page_number = no_doc_files;
        } else {
            sprintf(seg->segment_url, "%s_%d.html",
                read_placeholder("SOURCEPREFIX"), no_src_files++);
            seg->page_number = no_src_files;
        }
    }
```

This code is used in §20.

§**22.**

⟨Work out how the segments link together 22⟩ ≡

```
segment *seg, *first_doc_seg = NULL, *first_src_seg = NULL;
LOOP_OVER(seg, segment) {
    if (seg->documentation) {
        seg->link_home = NULL;
        seg->link_contents = NULL;
        seg->link_previous = NULL;
        seg->link_next = NULL;
        if (first_doc_seg == NULL) first_doc_seg = seg;
    } else {
        seg->link_home = NULL;
        seg->link_contents = NULL;
        seg->link_previous = NULL;
        seg->link_next = NULL;
        if (first_src_seg == NULL) {
            first_src_seg = seg;
            seg->link_previous = contents_leafname;
        }
    }
}
LOOP_OVER(seg, segment) {
    if (seg->documentation) {
        seg->link_home = "index.html";
        seg->link_contents = first_doc_seg->segment_url;
    } else {
        seg->link_home = "index.html";
        seg->link_contents = contents_leafname;
    }
    segment *before = seg;
    while (TRUE) {
        before = PREV_OBJECT(before, segment);
        if (before == NULL) break;
        if (before->documentation == seg->documentation) {
            seg->link_previous = before->segment_url; break;
        }
    }
    segment *after = seg;
    while (TRUE) {
        after = NEXT_OBJECT(after, segment);
        if (after == NULL) break;
        if (after->documentation == seg->documentation) {
            seg->link_next = after->segment_url; break;
        }
    }
}
```

This code is used in §20.

§23.

⟨Generate the prefatory page, which isn't a segment 23⟩ ≡

```
segment_being_written = NULL;
source_HTML_pages_created++;
web_copy(template, contents_page);
```

This code is used in §20.

§24.

⟨Generate the segment pages 24⟩ ≡

```
segment *seg;
LOOP_OVER(seg, segment) {
    char segment_page[MAX_FILENAME_LENGTH];
    sprintf(segment_page, "%s%c%s", website_pathname, SEP_CHAR, seg->segment_url);
    segment_being_written = seg;
    source_HTML_pages_created++;
    web_copy(template, segment_page);
    segment_being_written = NULL;
}
```

This code is used in §20.

§25. This is what "[PAGENUMBER]" in the template becomes.

```
void expand_PAGENUMBER_variable(FILE *COPYTO) {
    int p = 1;
    if (segment_being_written) {
        p = segment_being_written->page_number;
        if (segment_being_written->documentation == FALSE) p++;                    allow for header page
    }
    fprintf(COPYTO, "%d", p);
}
```

The function expand_PAGENUMBER_variable is called from 3/place.

§26. And similarly "[PAGEEXTENT]".

```
void expand_PAGEEXTENT_variable(FILE *COPYTO) {
    int n = no_src_files + 1;
    if ((segment_being_written) && (segment_being_written->documentation))
        n = no_doc_files;
    if (n == 0) n = 1;
    fprintf(COPYTO, "%d", n);
}
```

The function expand_PAGEEXTENT_variable is called from 3/place.

§**27.** And this is what "[SOURCELINKS]" in the template becomes:

```
void expand_SOURCELINKS_variable(FILE *COPYTO) {
    segment *seg = segment_being_written;
    if (seg) {
        if (seg->link_home)
            fprintf(COPYTO, "<li><a href=\"%s\">Home page</a></li>", seg->link_home);
        if (seg->link_contents)
            fprintf(COPYTO, "<li><a href=\"%s\">Beginning</a></li>", seg->link_contents);
        if (seg->link_previous)
            fprintf(COPYTO, "<li><a href=\"%s\">Previous</a></li>", seg->link_previous);
        if (seg->link_next)
            fprintf(COPYTO, "<li><a href=\"%s\">Next</a></li>", seg->link_next);
    } else {
        fprintf(COPYTO, "<li><a href=\"index.html\">Home page</a></li>");
        fprintf(COPYTO, "<li><a href=\"%s.txt\">Complete text</a></li>",
            read_placeholder("SOURCEPREFIX"));
    }
}
```

The function expand_SOURCELINKS_variable is called from 3/place.

§**28.** When working on "[SOURCE]" or "[SOURCENOTES]", we will need to run through a segment of the source text, one line at a time. As we do so, we'll maintain the following variables, along with `current_style` (for which see the CSS discussion above):

```
FILE *SPAGE = NULL;                          where the output is going
int SOURCENOTES_mode = FALSE;                TRUE for "[SOURCENOTES]", FALSE for "[SOURCE]"
int quoted_matter = FALSE;                   are we inside double-quoted matter in the source text?
int i6_matter = FALSE;                       are we inside verbatim I6 code in the source text?
int comment_nesting = 0;                     nesting level of comments in source text being read: 0 for not in a comment
int carry_over_indentation = -1;             indentation carried over for para breaks in quoted text
int next_footnote_number = 1;                number to assign to the next footnote which comes up
heading *latest_heading = NULL;              a heading which is always behind the current position
table *latest_table = NULL;                  a table which is always behind the current position
```

§**29.** So this is "[SOURCE]" (if `noting_mode` is `FALSE`) or "[SOURCENOTES]" (if `TRUE`).

```
void expand_SOURCE_or_SOURCENOTES_variable(FILE *write_to, int SN) {
    if (SN) ⟨Typeset the little Notes subheading 31⟩;
    open_code(write_to);
    ⟨Initialise the variables to their state at the start of an HTML page 30⟩;
    ⟨Read the source text and feed it one line at a time to the line-writer 32⟩;
    close_code(write_to);
}
```

The function expand_SOURCE_or_SOURCENOTES_variable is called from 3/place.

§**30.** So at the start of the preface or of any segment:

⟨Initialise the variables to their state at the start of an HTML page 30⟩ ≡

```
next_footnote_number = 1;
SPAGE = write_to;
SOURCENOTES_mode = SN;
quoted_matter = FALSE;
i6_matter = FALSE;
comment_nesting = 0;
carry_over_indentation = -1;
current_style = NULL;
latest_heading = FIRST_OBJECT(heading);
latest_table = FIRST_OBJECT(table);
```

This code is used in §29.

§**31.** We expect any use of "[SOURCENOTES]" to come after the relevant "[SOURCE]", so that looking at `next_footnote_number` will tell us how many notes there were.

⟨Typeset the little Notes subheading 31⟩ ≡

```
if (next_footnote_number == 1) return;                          there were no footnotes at all
fprintf(write_to, "<p>");
open_style(write_to, "notesheading");
if (next_footnote_number == 2) fprintf(write_to, "Note");                                  just one
else fprintf(write_to, "Notes");                                                      more than one
close_style(write_to, "notesheading");
fprintf(write_to, "</p>\n");
```

This code is used in §29.

§**32.** We want to be very careful about running time here. This paragraph will run about H times, where H is the number of headings (in fact at most $H+1$ times and usually a little less); but we might reasonably expect that H is proportional to N, since there's typically a heading every 30 or so lines in the source text, so that $H \simeq N/30$. If we then did the simplest thing, of opening the source text file and sending every line to `write_source_line`, we would make $O(N^2)$ calls, and even though many of those would quickly return it would be an expensive algorithm.

Instead, we start at the relevant position in the source text for the current HTML page, and we stop the moment that `write_source_line` reports that it has gone past the material of interest. We thus make at most $N+H$ calls to `write_source_line` (the extra H calls being for one overspill line per segment, where we realise that we've gone too far).

⟨Read the source text and feed it one line at a time to the line-writer 32⟩ ≡

```
text_file_position *start = NULL;
if (segment_being_written) ⟨Start from just the right place in the source file 33⟩;
file_read(source_text, "can't open source text", TRUE, source_write_iterator, start);
```

This code is used in §29.

§**33.** The following simulates the effect of running through the uninteresting lines before the segment begins:

⟨Start from just the right place in the source file 33⟩ ≡
```
start = &(segment_being_written->start_position_in_file);
if (segment_being_written->most_recent_heading)
    latest_heading = segment_being_written->most_recent_heading;
if (segment_being_written->most_recent_table)
    latest_table = segment_being_written->most_recent_table;
```

This code is used in §32.

§**34.**

```
void source_write_iterator(char *line, text_file_position *tfp) {
    int done_yet = write_source_line(line, tfp);
    if (done_yet) tfp_lose_interest(tfp);
}
```

The function source_write_iterator is.

§**35.** And this is where we write lines. We arrive here with exactly the same line count as the scanner observed before on pass 1, so we can validly compare our current line count against those stored for tables, headings and segments.

When this routine returns `TRUE`, it signals that there is no further need for the source text, and that saves reading in all of the remaining lines which won't be needed.

```
int write_source_line(char *line, text_file_position *tfp) {
    int line_count = tfp_get_line_count(tfp);
    if (segment_being_written == NULL) ⟨Filter out lines for the preface 36⟩
    else ⟨Filter out lines for the segments 37⟩;
    if (SOURCENOTES_mode) ⟨Typeset the line in [SOURCENOTES] mode 38⟩
    else ⟨Typeset the line in [SOURCE] mode 39⟩;
    return FALSE;
}
```

The function write_source_line is.

§**36.** Recall that the source text is divided into an initial portion containing no headings – the "preface" – and then segments, each of which begins with a heading.

Here we are handling the case of typesetting the preface. We allow the line to appear as normal if it is before the first segment; once we reach the first segment – if there's a first segment to reach – we then typeset the contents listing. (If there's no first segment, then there are no headings, and there's no need for a contents listing.) If we've output the contents listing then we are finished writing the preface and don't need to read the source text further, so we return `TRUE`.

⟨Filter out lines for the preface 36⟩ ≡
```
segment *first_segment = FIRST_OBJECT(segment);
if ((first_segment) && (line_count == first_segment->begins_at - 1) && (line[0] == 0))
    return FALSE;                 don't bother to typeset a blank line just before the first segment is reached
if ((first_segment) && (line_count == first_segment->begins_at)) {
    if (SOURCENOTES_mode == FALSE) typeset_contents_listing(TRUE);
    return TRUE;
}
```

This code is used in §35.

§**37.** The segment pages are easier: in this case we allow the line only if it lies inside the segment, and otherwise suppress it. Once we've gone beyond the segment, we don't need to read any further, so we return `TRUE`.

⟨Filter out lines for the segments 37⟩ ≡

```
if (line_count < segment_being_written->begins_at) return FALSE;
if (line_count > segment_being_written->ends_at) return TRUE;
if (line_count == position_of_documentation_bar + 1)
    typeset_contents_listing(FALSE);
```

This code is used in §35.

§**38.** In [SOURCENOTES] mode, we detect footnotes in the form of comments in the source text marked by asterisks; each one is assigned the next footnote number, and typeset. All other material is ignored.

⟨Typeset the line in [SOURCENOTES] mode 38⟩ ≡

```
int i;
for (i=0; line[i]; i++) {
    if ((line[i] == '[') && (line[i+1] == '*')) {
        int comment_level = 1;
        fprintf(SPAGE, "<p><a name=\"note%d\"></a>", next_footnote_number);
        open_style(SPAGE, "notetext");
        fprintf(SPAGE, "<a href=\"#note%dref\">[%d]</a>. ",
            next_footnote_number, next_footnote_number);
        next_footnote_number++;
        i+=2;
        while (line[i]) {
            if (line[i] == '[') comment_level++;
            if (line[i] == ']') comment_level--;
            if (comment_level == 0) break;
            fprintf(SPAGE, "%c", line[i++]);
        }
        close_style(SPAGE, "notetext");
        fprintf(SPAGE, "</p>\n");
    }
}
```

This code is used in §35.

§**39.** In [SOURCE] mode, we need to work out appropriate type styles to embellish the line, then indent it suitably, then typeset it character by character.

⟨Typeset the line in [SOURCE] mode 39⟩ ≡

```
int embolden = FALSE, tabulate = FALSE, underline = FALSE;
⟨Decide any typographic embellishments due to the line falling inside a table 42⟩;
⟨The top line of the preface or any segment is in bold 43⟩;
⟨Any heading line is in bold 44⟩;
if ((tabulate) && (quoted_matter == FALSE)) { fprintf(SPAGE, "<tr>"); open_table_cell(SPAGE); }
int start = 0;
if (tabulate == FALSE) {
    for (; line[start] == '\t'; start++) ;
    if (carry_over_indentation < 0) carry_over_indentation = start;
    open_code_paragraph(SPAGE, carry_over_indentation);
}
⟨Begin typographic embellishments 40⟩;
⟨The documentation requires some corrections 45⟩;
int i; for (i=start; line[i]; i++) ⟨Typeset a single character of the source text 46⟩;
⟨End typographic embellishments 41⟩;
if ((tabulate) && (quoted_matter == FALSE)) { close_table_cell(SPAGE); fprintf(SPAGE, "</tr>\n");
}
else close_code_paragraph(SPAGE);
if (quoted_matter == FALSE) carry_over_indentation = -1;
```

This code is used in §35.

§**40.** The type styles are easily applied, so let's do that now. The innermost one must be colour, since that may change in the course of the line.

⟨Begin typographic embellishments 40⟩ ≡

```
if (underline) open_style(SPAGE, "columnhead");
if (embolden) open_style(SPAGE, "heading");
if (current_style) open_style(SPAGE, current_style);
```

This code is used in §39.

§**41.** And they end in reverse order, so that they nest properly if need be:

⟨End typographic embellishments 41⟩ ≡

```
if (current_style) close_style(SPAGE, current_style);
if (embolden) close_style(SPAGE, "heading");
if (underline) close_style(SPAGE, "columnhead");
```

This code is used in §39.

§**42.** The heading line of a source text Table is in bold; the column-headings line is underlined; and the material inside appears in an HTML table, with `tabulate` mode set.

The `while` loop here needs a careful look, since on the face of it this could mean $O(N)$ iterations – since the number of tables is probably proportional to N – made in the course of the current "[SOURCE]" expansion. Since the number of "[SOURCE]" expansions needed to make the website is also $O(N)$ – the number of HTML pages in the site is proportional to the number of headings, which is also proportional to N – there's a risk that this `while` loop makes the whole website algorithm $O(N^2)$. This is why, on each "[SOURCE]" expansion, `latest_table` is initialised not to the first table but to the most recent one at the start position of the current HTML page. Moreover, the loop never goes past the current line count, which never goes outside the range of lines in the current HTML page. The result is that over the course of all the "[SOURCE]" expansions combined, the `while` loop here executes $O(N)$ iterations in total.

⟨Decide any typographic embellishments due to the line falling inside a table 42⟩ ≡

```
while ((latest_table) && (latest_table->table_line_end < line_count))
    latest_table = NEXT_OBJECT(latest_table, table);
if (latest_table) {
    int from = latest_table->table_line_start, to = latest_table->table_line_end;
    if (line_count == from) {
        embolden = TRUE;
    } else if ((line_count > from) && (line_count < to)) {
        tabulate = TRUE;
        if (line_count == from + 1) {
            underline = TRUE;
            fprintf(SPAGE, "<table>");
        }
    } else if (line_count == to) {
        fprintf(SPAGE, "</table>");
    }
}
```

This code is used in §39.

§**43.**

⟨The top line of the preface or any segment is in bold 43⟩ ≡

```
if ((line_count == 1) ||
    ((segment_being_written) && (line_count == segment_being_written->begins_at)))
        embolden = TRUE;
```

This code is used in §39.

§**44.** See the discussion of `latest_table` above for why the following `while` loop also doesn't make our algorithm $O(N^2)$.

⟨Any heading line is in bold 44⟩ ≡

```
while ((latest_heading) && (latest_heading->heading_line < line_count))
    latest_heading = NEXT_OBJECT(latest_heading, heading);
if ((latest_heading) && (latest_heading->heading_line == line_count))
    embolden = TRUE;
```

This code is used in §39.

§**45.**

⟨The documentation requires some corrections 45⟩ ≡
```
if ((comment_nesting == 0) && (quoted_matter == FALSE) && (i6_matter == FALSE) &&
    (line[start] == '*') && (line[start+1] == ':') && (line[start+2] == ' '))
    start += 3;
if (line_count == position_of_documentation_bar) strcpy(line, "Documentation");
```

This code is used in §39.

§**46.** We need to do two things: ensure that the character is HTML-safe, which means escaping out `"`, `<`, `>` and `&` (but nothing else since the HTML file will use a UTF-8 encoding, the same as that in the source text); and keep track of the opening and closing of comments and quoted matter.

⟨Typeset a single character of the source text 46⟩ ≡
```
switch (line[i]) {
    case '\t':
        a multiple tab is equivalent to a single tab in Inform source text
        while (line[i+1] == '\t') i++;
        ⟨Typeset a tab 47⟩;
        break;
    case '"':
        if ((comment_nesting > 0) || (i6_matter)) fprintf(SPAGE, "&quot;");
        else ⟨Typeset a double quotation mark outside of a comment 48⟩;
        break;
    case '[':
        if (quoted_matter) { fprintf(SPAGE, "["); change_style(SPAGE, "substitution"); }
        else if (i6_matter) fprintf(SPAGE, "[");
        else ⟨Typeset an open square bracket outside of a string 49⟩;
        break;
    case ']':
        if (quoted_matter) { change_style(SPAGE, "quote"); fprintf(SPAGE, "]"); }
        else if (i6_matter) fprintf(SPAGE, "]");
        else ⟨Typeset a close square bracket outside of a string 50⟩;
        break;
    case '(':
        if ((comment_nesting == 0) && (quoted_matter == FALSE) && (i6_matter == FALSE) &&
            (line[i+1] == '-')) { i++;
            ⟨Typeset the opening of I6 verbatim code 51⟩
        } else fprintf(SPAGE, "("); break;
    case '-':
        if ((i6_matter) && (line[i+1] == ')')) { i++;
            ⟨Typeset the closing of I6 verbatim code 52⟩
        } else fprintf(SPAGE, "-"); break;
    case '<': fprintf(SPAGE, "&lt;"); break;
    case '>': fprintf(SPAGE, "&gt;"); break;
    case '&': fprintf(SPAGE, "&amp;"); break;
    default: fprintf(SPAGE, "%c", line[i]); break;
}
```

This code is used in §39.

§**47.** Inside a source-text Table, a tab moves to the next column, so we need to typeset a cell boundary in our HTML `<table>`. Outside of that context, a tab is just white space and we turn it into a single space.

⟨Typeset a tab 47⟩ ≡
```
if (tabulate) {
    ⟨End typographic embellishments 41⟩;
    close_table_cell(SPAGE);
    open_table_cell(SPAGE);
    ⟨Begin typographic embellishments 40⟩;
} else {
    fprintf(SPAGE, " ");
}
```
This code is used in §46.

§**48.** The following enters or exits quoted-matter mode, and is structured so that the quotation marks are not coloured – only the material inside them.

Our code in handling quoted and comment matter is greatly simplified by the fact that a valid Inform text cannot contain mismatched square brackets; however, as Dave Chapeskie points out, a valid comment can contain mismatched quotation marks, and this section of code benefits from his careful amendments.

⟨Typeset a double quotation mark outside of a comment 48⟩ ≡
```
if (quoted_matter) change_style(SPAGE, NULL);
fprintf(SPAGE, "&quot;");
if (quoted_matter == FALSE) change_style(SPAGE, "quote");
quoted_matter = (quoted_matter)?FALSE:TRUE;
```
This code is used in §46.

§**49.** On the other hand, the squares around a comment *do* pick up the colour of the commentary within them. Asterisked comments must end in the same paragraph as they begin.

⟨Typeset an open square bracket outside of a string 49⟩ ≡
```
if (line[i+1] == '*') {
    advance past the end of the asterisked comment
    int comment_level = 1;
    for (i+=2; line[i]; ++i) {
        if (line[i] == '[') comment_level++;
        if (line[i] == ']') comment_level--;
        if (comment_level == 0) break;
    }
    ⟨Typeset a footnote cue 53⟩;
} else {
    comment_nesting++;
    if (comment_nesting == 1) change_style(SPAGE, "comment");
    fprintf(SPAGE, "[");
}
```
This code is used in §46.

§**50.**

⟨Typeset a close square bracket outside of a string 50⟩ ≡

```
fprintf(SPAGE, "]");
comment_nesting--;
if (comment_nesting == 0) change_style(SPAGE, NULL);
```

This code is used in §46.

§**51.** Styling applied to I6 verbatim code does not apply to the purely-I7 markers "(-" and "-)" around it:

⟨Typeset the opening of I6 verbatim code 51⟩ ≡

```
fprintf(SPAGE, "(-");
change_style(SPAGE, "i6code");
i6_matter = TRUE;
```

This code is used in §46.

§**52.**

⟨Typeset the closing of I6 verbatim code 52⟩ ≡

```
change_style(SPAGE, NULL);
fprintf(SPAGE, "-)");
i6_matter = FALSE;
```

This code is used in §46.

§**53.** The "cue" of a footnote is the reference in the body of the text, which is conventionally printed as a superscript number. We leave that to the span `notecue` if we have CSS, and otherwise render in grey superscript.

⟨Typeset a footnote cue 53⟩ ≡

```
fprintf(SPAGE, "<a name=\"note%dref\"></a>", next_footnote_number);
open_style(SPAGE, "notecue");
fprintf(SPAGE, "<a href=\"#note%d\">[%d]</a>",
    next_footnote_number, next_footnote_number);
close_style(SPAGE, "notecue");
next_footnote_number++;
```

This code is used in §49.

§**54.** That just leaves the little contents listings – one for the source, and another for the documentation (if any).

```
void typeset_contents_listing(int source_contents) {
    int benchmark_level = (source_contents)?0:DOC_CHAPTER_LEVEL;
    int current_level = benchmark_level-1, new_level;
    heading *h;
    LOOP_OVER(h, heading)
        if (((source_contents) && (h->heading_line < position_of_documentation_bar)) ||
            ((source_contents == FALSE) && (h->heading_line > position_of_documentation_bar))) {
            new_level = h->heading_level;
            if (h->heading_level == EXAMPLE_LEVEL) new_level = DOC_CHAPTER_LEVEL;
            ⟨Open or close UL tags to move to the new heading level 55⟩;
            fprintf(SPAGE, "<li><a href=%s>%s</a></li>\n",
                h->heading_to_segment->segment_url, h->heading_text);
        }
    new_level = benchmark_level-1;
    ⟨Open or close UL tags to move to the new heading level 55⟩;
}
```

The function typeset_contents_listing is.

§**55.** This is how we obtain our nested UL tags: `current_level` starts and ends at $b-1$, and can only change its value by executing the following loops. Since it never changes to a value lower than 0 except when returning to $b-1$ at the end, we are always inside at least the outermost `<ul>`, and since the net change over the whole process is 0, there must be as many steps upward as downward – so every `<ul>` is closed by a matching `</ul>`.

⟨Open or close UL tags to move to the new heading level 55⟩ ≡

```
    while (new_level > current_level) { fprintf(SPAGE, "<ul>"); current_level++; }
    while (new_level < current_level) { fprintf(SPAGE, "</ul>"); current_level--; }
```

This code is used in §54.