# CBLORB

The Program

# Chapter 3

Build 3/100422    Graham Nelson

`cblorb` *is a command-line tool which forms one of the components of the Inform 7 design system for interactive fiction. All installations of Inform 7 contain a copy of* `cblorb`*, though few users are aware of it, since it doesn't usually communicate with them directly. Instead, the Inform user interface calls it when needed. The moment comes at the end of the translation process, but only when the Release button rather than the Go or Replay buttons was clicked.* `cblorb` *has two main jobs: to bind up the translated project, together with any pictures, sounds, or cover art, into a single file called a "blorb" which can be given to players on other machines to play; and to produce associated websites, solution files and so on as demanded by "Release..." instruction(s) in the source text.*

# 3 Other Material

# Releaser 3/rel

*Purpose*

To manage requests to release material other than a Blorb file.

*Definitions*

**¶1.** If the previous section, "Blorb Writer.w", was the Lord High Executioner, then this one is the Lord High Everything Else: it keeps track of requests to write all kinds of interesting things which are *not* blorb files, and then sees that they are carried out. The requests divide as follows:

```
define COPY_REQ 0                                    a miscellaneous file
define IFICTION_REQ 1                                the iFiction record of a project
define RELEASE_FILE_REQ 2                            a template file
define RELEASE_SOURCE_REQ 3                          the source text in HTML form
define SOLUTION_REQ 4                                a solution file generated from the skein
define SOURCE_REQ 5                                  the source text of a project
define WEBSITE_REQ 6                                 a whole website
define INTERPRETER_REQ 7                             an in-browser interpreter
define BASE64_REQ 8                                  a base64-encoded copy of a binary file
define INSTRUCTION_REQ 9                             a release instruction copied to cblorb for reporting only
define ALTERNATIVE_REQ 10                            an unused release instruction copied to cblorb for reporting only

int website_requested = FALSE;                       has a WEBSITE_REQ been made?
```

**¶2.** This would use a lot of memory if there were many requests, but there are not and it does not.

```
typedef struct request {
    int what_is_requested;                           one of the *_REQ values above
    char details1[MAX_FILENAME_LENGTH];
    char details2[MAX_FILENAME_LENGTH];
    char details3[MAX_FILENAME_LENGTH];
    int private;                                     is this request private, i.e., not to contribute to a website?
    int outcome_data;                                e.g. number of bytes copied
    MEMORY_MANAGEMENT
} request;
```

The structure request is private to this section.

§**1. Receiving requests.** These can have from 0 to 3 textual details attached:

```
request *request_0(int kind, int privacy) {
    request *req = CREATE(request);
    req->what_is_requested = kind;
    req->details1[0] = 0;
    req->details2[0] = 0;
    req->details3[0] = 0;
    req->private = privacy;
    req->outcome_data = 0;
    if (kind == WEBSITE_REQ) website_requested = TRUE;
    return req;
}
request *request_1(int kind, char *text1, int privacy) {
    request *req = request_0(kind, privacy);
    strcpy(req->details1, text1);
    return req;
}
request *request_2(int kind, char *text1, char *text2, int privacy) {
    request *req = request_0(kind, privacy);
    strcpy(req->details1, text1);
    strcpy(req->details2, text2);
    return req;
}
request *request_3(int kind, char *text1, char *text2, char *text3, int privacy) {
    request *req = request_0(kind, privacy);
    strcpy(req->details1, text1);
    strcpy(req->details2, text2);
    strcpy(req->details3, text3);
    return req;
}
```

The function request_0 is.

The function request_1 is.

The function request_2 is.

The function request_3 is.

§**2.** A convenient abbreviation:

```
void request_copy(char *from, char *to) {
    request_2(COPY_REQ, from, to, FALSE);
}
```

The function request_copy is called from 3/links.

**§3. Any Last Requests.** Most of the requests are made as the parser reads commands from the blurb script. At the end of that process, though, the following routine may add further requests as consequences:

```
void any_last_requests(void) {
    request_copy_of_auxiliaries();
    if (default_cover_used == FALSE) {
        char *BIGCOVER = read_placeholder("BIGCOVER");
        if (BIGCOVER) {
            if (cover_is_in_JPEG_format) request_copy(BIGCOVER, "Cover.jpg");
            else request_copy(BIGCOVER, "Cover.png");
        }
        if (website_requested) {
            char *SMALLCOVER = read_placeholder("SMALLCOVER");
            if (SMALLCOVER) {
                if (cover_is_in_JPEG_format) request_copy(SMALLCOVER, "Small Cover.jpg");
                else request_copy(SMALLCOVER, "Small Cover.png");
            }
        }
    }
}
```

The function any_last_requests is.

## §4. Carrying out requests.

```
void create_requested_material(void) {
    if (release_folder[0] == 0) return;
    printf("! Release folder: <%s>\n", release_folder);
    if (blorb_file_size > 0) declare_where_blorb_should_be_copied(release_folder);
    any_last_requests();
    request *req;
    LOOP_OVER(req, request) {
        switch (req->what_is_requested) {
            case ALTERNATIVE_REQ: break;
            case BASE64_REQ: ⟨Copy a base64-encoded file across 9⟩; break;
            case COPY_REQ: ⟨Copy a file into the release folder 8⟩; break;
            case IFICTION_REQ: ⟨Create an iFiction file 7⟩; break;
            case INSTRUCTION_REQ: break;
            case INTERPRETER_REQ: ⟨Create an in-browser interpreter 12⟩; break;
            case RELEASE_FILE_REQ: ⟨Release a file into the release folder 10⟩; break;
            case RELEASE_SOURCE_REQ: ⟨Release source text as HTML into the release folder 11⟩; break;
            case SOLUTION_REQ: ⟨Create a walkthrough file 5⟩; break;
            case SOURCE_REQ: ⟨Create a plain text source file 6⟩; break;
            case WEBSITE_REQ: ⟨Create a website 13⟩; break;
        }
    }
}
```

The function create_requested_material is called from 1/main.

**§5.**

⟨Create a walkthrough file 5⟩ ≡

```
char Skein_filename[MAX_FILENAME_LENGTH];
sprintf(Skein_filename, "%s%cSkein.skein", project_folder, SEP_CHAR);
char solution_filename[MAX_FILENAME_LENGTH];
sprintf(solution_filename, "%s%csolution.txt", release_folder, SEP_CHAR);
walkthrough(Skein_filename, solution_filename);
```

This code is used in §4.

**§6.**

⟨Create a plain text source file 6⟩ ≡

```
char source_text_filename[MAX_FILENAME_LENGTH];
sprintf(source_text_filename, "%s%cSource%cstory.ni",
    project_folder, SEP_CHAR, SEP_CHAR);
char write_to[MAX_FILENAME_LENGTH];
sprintf(write_to, "%s%csource.txt", release_folder, SEP_CHAR);
copy_file(source_text_filename, write_to, FALSE);
```

This code is used in §4.

**§7.**

⟨Create an iFiction file 7⟩ ≡

```
char iFiction_filename[MAX_FILENAME_LENGTH];
sprintf(iFiction_filename, "%s%cMetadata.iFiction", project_folder, SEP_CHAR);
char write_to[MAX_FILENAME_LENGTH];
sprintf(write_to, "%s%ciFiction.xml", release_folder, SEP_CHAR);
copy_file(iFiction_filename, write_to, FALSE);
```

This code is used in §4.

**§8.**

⟨Copy a file into the release folder 8⟩ ≡

```
char write_to[MAX_FILENAME_LENGTH];
sprintf(write_to, "%s%c%s", release_folder, SEP_CHAR, req->details2);
int size = copy_file(req->details1, write_to, TRUE);
req->outcome_data = size;
if (size == -1) {
    int i;
    for (i = strlen(req->details1); i>0; i--)
        if ((req->details1)[i] == SEP_CHAR) { i++; break; }
    errorf_1s(
        "You asked to release along with a file called '%s', which ought "
        "to be in the Materials folder for the project. But I can't find "
        "it there.", (req->details1)+i);
}
```

This code is used in §4.

**§9.**

⟨Copy a base64-encoded file across 9⟩ ≡

```
    encode_as_base64(req->details1, req->details2,
        read_placeholder("BASESIXTYFOURTOP"), read_placeholder("BASESIXTYFOURTAIL"));
```

This code is used in §4.

**§10.**

⟨Release a file into the release folder 10⟩ ≡

```
    release_file_into_website(req->details1, req->details2, NULL);
```

This code is used in §4.

**§11.**

⟨Release source text as HTML into the release folder 11⟩ ≡

```
    set_placeholder_to("SOURCEPREFIX", "source", 0);
    set_placeholder_to("SOURCELOCATION", req->details1, 0);
    set_placeholder_to("TEMPLATE", req->details3, 0);
    char *HTML_template = find_file_in_named_template(req->details3, req->details2);
    if (HTML_template == NULL) error_1("can't find HTML template file", req->details2);
    if (trace_mode) printf("! Web page %s from template %s\n", HTML_template, req->details3);
    web_copy_source(HTML_template, release_folder);
```

This code is used in §4.

**§12.** Interpreters are copied, not made. They're really just like website templates, except that they have a manifest file instead of an extras file, and that they're copied into an `interpreter` subfolder of the release folder, which is assumed already to exist. (It isn't copied because folder creation is tiresome to do in a cross-platform way, since Windows doesn't follow POSIX. The necessary code exists in Inform already, so we'll do it there.)

⟨Create an in-browser interpreter 12⟩ ≡

```
    set_placeholder_to("INTERPRETER", req->details1, 0);
    char *t = read_placeholder("INTERPRETER");
    char *from = find_file_in_named_template(t, "(manifest).txt");
    if (from) {                                  i.e., if the "(manifest).txt" file exists
        file_read(from, "can't open (manifest) file", FALSE, read_requested_ifile, 0);
    }
```

This code is used in §4.

**§13.** We copy the CSS file, if we need one; make the home page; and make any other pages demanded by public released material. After that, it's up to the template to add more if it wants to.

```
⟨Create a website 13⟩ ≡
    set_placeholder_to("TEMPLATE", req->details1, 0);
    char *t = read_placeholder("TEMPLATE");
    if (use_css_code_styles) {
        char *from = find_file_in_named_template(t, "style.css");
        if (from) {
            char CSS_filename[MAX_FILENAME_LENGTH];
            sprintf(CSS_filename, "%s%cstyle.css", release_folder, SEP_CHAR);
            copy_file(from, CSS_filename, FALSE);
        }
    }
    release_file_into_website("index.html", t, NULL);
    request *req;
    LOOP_OVER(req, request)
        if (req->private == FALSE)
            switch (req->what_is_requested) {
                case INTERPRETER_REQ:
                    release_file_into_website("play.html", t, NULL); break;
                case SOURCE_REQ:
                    set_placeholder_to("SOURCEPREFIX", "source", 0);
                        char source_text[MAX_FILENAME_LENGTH];
                    sprintf(source_text, "%s%cSource%cstory.ni",
                        project_folder, SEP_CHAR, SEP_CHAR);
                    set_placeholder_to("SOURCELOCATION", source_text, 0);
                    release_file_into_website("source.html", t, NULL); break;
            }
    ⟨Add further material as requested by the template 14⟩;
```

This code is used in §4.

**§14.** Most templates do not request extra files, but they have the option by including a manifest called "(extras).txt":

```
⟨Add further material as requested by the template 14⟩ ≡
    char *from = find_file_in_named_template(t, "(extras).txt");
    if (from) {                                   i.e., if the "(extras).txt" file exists
        file_read(from, "can't open (extras) file", FALSE, read_requested_file, 0);
    }
```

This code is used in §13.

**§15. The Extras file for a website template.** When parsing "(extras).txt", read_requested_file is called for each line. We trim white space and expect the result to be a filename of something within the template.

```
void read_requested_file(char *filename, text_file_position *tfp) {
    filename = trim_white_space(filename);
    if (filename[0] == 0) return;
    release_file_into_website(filename, read_placeholder("TEMPLATE"), NULL);
}
```

The function read_requested_file is.

**§16. The Manifest file for an interpreter.** When parsing "(manifest).txt", we do almost the same thing. Like a website template, an interpreter is stored in a single folder, and the manifest can list files which need to be copied into the Release in order to piece together a working copy of the interpreter.

However, this is more expressive than the "(extras).txt" file because it also has the ability to set placeholders in `cblorb`. We use this mechanism because it allows each interpreter to provide some metadata about its own identity and exactly how it wants to be interfaced with the website which `cblorb` will generate. This isn't the place to document what those metadata placeholders are and what they mean, since (except for a consistency check below) `cblorb` doesn't know anything about them – it's the Standard website template which they need to match up to. Anyway, the best way to get an idea of this is to read the manifest file for the default, Parchment, interpreter.

```
char current_placeholder[MAX_VAR_NAME_LENGTH];
int cp_written = FALSE;
void read_requested_ifile(char *manifestline, text_file_position *tfp) {
    if (cp_written == FALSE) { cp_written = TRUE; current_placeholder[0] = 0; }
    manifestline = trim_white_space(manifestline);
    if (manifestline[0] == '[') ⟨Go into or out of placeholder setting mode 17⟩;
    if (current_placeholder[0] == 0)
        ⟨We're outside placeholder mode, so it's a comment or a manifested filename 18⟩
    else
        ⟨We're inside placeholder mode, so it's content to be spooled into the named placeholder 19⟩;
}
```

The function read_requested_ifile is.

**§17.** Placeholders are set thus:

```
    [INTERPRETERVERSION]
    Parchment for Inform 7
    []
```

where the opening line names the placeholder, then one or more lines give the contents, and the box line ends the definition.

We're in the mode if `current_placeholder` is a non-empty C string, and if so, then it's the name of the one being set. Thus the code to handle the opening and closing lines can be identical.

```
⟨Go into or out of placeholder setting mode 17⟩ ≡
    if (manifestline[strlen(manifestline)-1] == ']') {
        if (strlen(manifestline) >= MAX_VAR_NAME_LENGTH) {
            error_1("placeholder name too long in manifest file", manifestline);
            return;
        }
        strcpy(current_placeholder, manifestline+1);
        current_placeholder[strlen(current_placeholder)-1] = 0;
        return;
    }
    error_1("placeholder name lacks ']' in manifest file", manifestline);
    return;
```

This code is used in §16.

**§18.** Outside of placeholders, blank lines and lines introduced by the comment character `!` are skipped.

⟨We're outside placeholder mode, so it's a comment or a manifested filename 18⟩ ≡

```
if ((manifestline[0] == '!') || (manifestline[0] == 0)) return;
release_file_into_website(manifestline, read_placeholder("INTERPRETER"), "interpreter");
```

This code is used in §16.

**§19.** Line breaks are included between lines, though not at the end of the final line, so that a one-line definition like the example above contains no line break. White space is stripped out at the left and right hand edges of each line.

⟨We're inside placeholder mode, so it's content to be spooled into the named placeholder 19⟩ ≡

```
if (strcmp(current_placeholder, "INTERPRETERVM") == 0)
    ⟨Check the value being given against the actual VM we're blorbing up 20⟩;
if (read_placeholder(current_placeholder))
    append_to_placeholder(current_placeholder, "\n");
append_to_placeholder(current_placeholder, manifestline);
```

This code is used in §16.

**§20.** Perhaps it's clumsy to do it here, but at some point `cblorb` needs to make sure we aren't trying to release a Z-machine game along with a Glulx interpreter, or vice versa. The manifest file for the interpreter is required to declare which virtual machines it implements, by giving a value of the placeholder `INTERPRETERVM`. This declares whether the interpreter can handle blorbed Z-machine files (`z`), blorbed Glulx files (`g`) or both (`zg` or `gz`). No other values are legal; note lower case. `cblorb` then checks this against its own placeholder `INTERPRETERVMIS`, which stores what the actual format of the blorb being released is.

⟨Check the value being given against the actual VM we're blorbing up 20⟩ ≡

```
char *vm_used = read_placeholder("INTERPRETERVMIS");
int i, capable = FALSE;
for (i=0; manifestline[i]; i++)
    if (vm_used[0] == manifestline[i]) capable = TRUE;
if (capable == FALSE) {
    char *format = "Z-machine";
    if (vm_used[0] == 'g') format = "Glulx";
    errorf_2s(
        "You asked to release along with a copy of the '%s' in-browser "
        "interpreter, but this can't handle story files which use the "
        "%s story file format. (The format can be changed on Inform's "
        "Settings panel for a project.)",
        read_placeholder("INTERPRETER"), format);
}
```

This code is used in §19.

**§21.** There are really three cases when we release something from a website template. We can copy it verbatim as a binary file, we can expand placeholders but otherwise copy as a single item, or we can use it to make a mass generation of source pages.

```
void release_file_into_website(char *name, char *t, char *sub) {
    char write_to[MAX_FILENAME_LENGTH];
    if (sub) sprintf(write_to, "%s%c%s%c%s",
        release_folder, SEP_CHAR, sub, SEP_CHAR, name);
    else sprintf(write_to, "%s%c%s", release_folder, SEP_CHAR, name);

    char *from = find_file_in_named_template(t, name);
    if (from == NULL) {
        error_1("unable to find file in website template", name);
        return;
    }

    if (strcmp(get_filename_extension(name), ".html") == 0)
        ⟨Release an HTML page from the template into the website 22⟩
    else
        ⟨Release a binary file from the template into the website 23⟩;
}
```

The function release_file_into_website is.

**§22.** "Source.html" is a special case, as it expands into a whole suite of pages automagically. Otherwise we work out the filenames and then hand over to the experts.

```
⟨Release an HTML page from the template into the website 22⟩ ≡
    set_placeholder_to("TEMPLATE", t, 0);
    if (trace_mode) printf("! Web page %s from template %s\n", name, t);
    if (strcmp(name, "source.html") == 0)
        web_copy_source(from, release_folder);
    else
        web_copy(from, write_to);
```

This code is used in §21.

**§23.**

```
⟨Release a binary file from the template into the website 23⟩ ≡
    if (trace_mode) printf("! Binary file %s from template %s\n", name, t);
    copy_file(from, write_to, FALSE);
```

This code is used in §21.

**§24.** The home page will need links to any public released resources, and this is where those are added (to the other links already present, that is).

```
void add_links_to_requested_resources(FILE *COPYTO) {
    request *req;
    LOOP_OVER(req, request)
        if (req->private == FALSE)
            switch (req->what_is_requested) {
                case WEBSITE_REQ: break;
                case INTERPRETER_REQ:
                    fprintf(COPYTO, "<li>");
                    download_link(COPYTO, "Play In-Browser", NULL, "play.html", "link");
                    fprintf(COPYTO, "</li>");
                    break;
                case SOURCE_REQ:
                    fprintf(COPYTO, "<li>");
                    download_link(COPYTO, "Source Text", NULL, "source.html", "link");
                    fprintf(COPYTO, "</li>");
                    break;
                case SOLUTION_REQ:
                    fprintf(COPYTO, "<li>");
                    download_link(COPYTO, "Solution", NULL, "solution.txt", "link");
                    fprintf(COPYTO, "</li>");
                    break;
                case IFICTION_REQ:
                    fprintf(COPYTO, "<li>");
                    download_link(COPYTO, "Library Card", NULL, "iFiction.xml", "link");
                    fprintf(COPYTO, "</li>");
                    break;
            }
}
```

The function add_links_to_requested_resources is called from 3/links.

**§25. Blorb relocation.** This is a little dodge used to make the process of releasing games in Inform 7 more seamless: see the manual for an explanation.

```
void declare_where_blorb_should_be_copied(char *path) {
    char *leaf = read_placeholder("STORYFILE");
    if (leaf == NULL) leaf = "Story";
    printf("Copy blorb to: [[%s%c%s]]\n", path, SEP_CHAR, leaf);
}
```

The function declare_where_blorb_should_be_copied is.

**§26. Reporting the release.** Inform normally asks `cblorb` to generate an HTML page reporting what it has done, and if things have gone well then this typically contains a list of what has been released. (That's easy for us to produce, since we just have to look through the requests.) Rather than attempt to write to the file here, we copy the necessary HTML into the placeholder `ph`.

```
void report_requested_material(char *ph) {
    if (release_folder[0] == 0) return;                    this should never happen
    int launch_website = FALSE, launch_play = FALSE;
    append_to_placeholder(ph, "<ul>");
    ⟨Itemise the blorb file, possibly mentioning pictures and sounds 27⟩;
    ⟨Itemise the website, mentioning how many pages it has 28⟩;
    ⟨Itemise the interpreter 29⟩;
    ⟨Itemise the library card 30⟩;
    ⟨Itemise the solution file 31⟩;
    ⟨Itemise the source text 32⟩;
    ⟨Itemise auxiliary files in a sub-list 33⟩;
    append_to_placeholder(ph, "</ul>");
    if ((launch_website) || (launch_play))
        ⟨Give a centred line of links to the main web pages produced 34⟩;
    ⟨Add in links to release instructions from Inform source text 35⟩;
    ⟨Add in advertisements for features Inform would like to offer 36⟩;
}
```

The function report_requested_material is called from 1/main.

**§27.**

```
⟨Itemise the blorb file, possibly mentioning pictures and sounds 27⟩ ≡
    if ((no_pictures_included > 1) || (no_sounds_included > 0))
        append_to_placeholder(ph,
            "<li>The blorb file <b>[STORYFILE]</b> ([BLORBFILESIZE]K in size, "
            "including [BLORBFILEPICTURES] figures(s) and [BLORBFILESOUNDS] "
            "sound(s))</li>");
    else
        append_to_placeholder(ph,
            "<li>The blorb file <b>[STORYFILE]</b> ([BLORBFILESIZE]K in size)</li>");
```

This code is used in §26.

**§28.**

```
⟨Itemise the website, mentioning how many pages it has 28⟩ ≡
    if (count_requests_of_type(WEBSITE_REQ) > 0) {
        append_to_placeholder(ph,
            "<li>A website (generated from the [TEMPLATE] template) of ");
        char pcount[32];
        sprintf(pcount, "%d page%s", HTML_pages_created, (HTML_pages_created!=1)?"s":"");
        append_to_placeholder(ph, pcount);
        append_to_placeholder(ph, "</li>");
        launch_website = TRUE;
    }
```

This code is used in §26.

## §29.

⟨Itemise the interpreter 29⟩ ≡

```
if (count_requests_of_type(INTERPRETER_REQ) > 0) {
    launch_play = TRUE;
    append_to_placeholder(ph,
        "<li>A play-in-browser page (generated from the [INTERPRETER] interpreter)</li>");
}
```

This code is used in §26.

## §30.

⟨Itemise the library card 30⟩ ≡

```
if (count_requests_of_type(IFICTION_REQ) > 0)
    append_to_placeholder(ph,
        "<li>The library card (stored as an iFiction record)</li>");
```

This code is used in §26.

## §31.

⟨Itemise the solution file 31⟩ ≡

```
if (count_requests_of_type(SOLUTION_REQ) > 0)
    append_to_placeholder(ph,
        "<li>A solution file</li>");
```

This code is used in §26.

## §32.

⟨Itemise the source text 32⟩ ≡

```
if (count_requests_of_type(SOURCE_REQ) > 0) {
    if (source_HTML_pages_created > 0) {
        append_to_placeholder(ph, "<li>The source text (as plain text and as ");
        char pcount[32];
        sprintf(pcount, "%d web page%s",
            source_HTML_pages_created, (source_HTML_pages_created!=1)?"s":"");
        append_to_placeholder(ph, pcount);
        append_to_placeholder(ph, ")</li>");
    }
}
if (count_requests_of_type(RELEASE_SOURCE_REQ) > 0)
    append_to_placeholder(ph,
        "<li>The source text (as part of the website)</li>");
```

This code is used in §26.

§**33.**

⟨Itemise auxiliary files in a sub-list 33⟩ ≡

```
if (count_requests_of_type(COPY_REQ) > 0) {
    append_to_placeholder(ph, "<li>The following additional file(s):<ul>");
    request *req;
    LOOP_OVER(req, request)
        if (req->what_is_requested == COPY_REQ) {
            char *leafname = req->details2;
            append_to_placeholder(ph, "<li>");
            append_to_placeholder(ph, leafname);
            if (req->outcome_data >= 4096) {
                char filesize[32];
                sprintf(filesize, " (%dK)", req->outcome_data/1024);
                append_to_placeholder(ph, filesize);
            } else if (req->outcome_data >= 0) {
                char filesize[32];
                sprintf(filesize, " (%d byte%s)",
                    req->outcome_data, (req->outcome_data!=1)?"s":"");
                append_to_placeholder(ph, filesize);
            }
            append_to_placeholder(ph, "</li>");
        }
    append_to_placeholder(ph, "</ul></li>");
}
```

This code is used in §26.

§**34.** These two links are handled by means of LAUNCH icons which, if clicked, open the relevant pages not in the Inform application but using an external web browser (e.g., Safari on most Mac OS X installations). We can only achieve this effect using a Javascript function provided by the Inform application, called `openUrl`.

⟨Give a centred line of links to the main web pages produced 34⟩ ≡

```
append_to_placeholder(ph, "<p><center>");
if (launch_website) {
    append_to_placeholder(ph,
        "<a href=\"[JAVASCRIPTPRELUDE]"
        "openUrl('file://[**MATERIALSFOLDERPATHOPEN]/Release/index.html')\">"
        "<img src='inform:/launch.png' border=0></a> home page");
}
if ((launch_website) && (launch_play))
    append_to_placeholder(ph, " : ");
if (launch_play) {
    append_to_placeholder(ph,
        "<a href=\"[JAVASCRIPTPRELUDE]"
        "openUrl('file://[**MATERIALSFOLDERPATHOPEN]/Release/play.html')\">"
        "<img src='inform:/launch.png' border=0></a> play-in-browser page");
}
append_to_placeholder(ph, "</center></p>");
```

This code is used in §26.

**§35.** Since `cblorb` has no knowledge of what the Inform source text producing this blorb was, it can't finish the status report from its own knowledge – it must rely on details supplied to it by Inform via blurb commands. First, Inform gives it source-text links for any "Release along with..." sentences, which have by now become `INSTRUCTION_REQ` requests:

⟨Add in links to release instructions from Inform source text 35⟩ ≡

```
    request *req;
    int count = 0;
    LOOP_OVER(req, request)
        if (req->what_is_requested == INSTRUCTION_REQ) {
            if (count == 0)
                append_to_placeholder(ph, "<p>The source text gives release instructions ");
            else
                append_to_placeholder(ph, " and ");
            append_to_placeholder(ph, req->details1);
            append_to_placeholder(ph, " here");
            count++;
        }
    if (count > 0)
        append_to_placeholder(ph, ".</p>");
```

This code is used in §26.

**§36.** And secondly, Inform gives it adverts for other fancy services on offer, complete with links to the Inform documentation (which, again, `cblorb` doesn't itself know about); and these have by now become `ALTERNATIVE_REQ` requests.

⟨Add in advertisements for features Inform would like to offer 36⟩ ≡

```
    request *req;
    int count = 0;
    LOOP_OVER(req, request)
        if (req->what_is_requested == ALTERNATIVE_REQ) {
            if (count == 0)
                append_to_placeholder(ph,
                    "<p>Here are some other possibilities you might want to consider:<p><ul>");
            append_to_placeholder(ph, "<li>");
            append_to_placeholder(ph, req->details1);
            append_to_placeholder(ph, "</li>");
            count++;
        }
    if (count > 0)
        append_to_placeholder(ph, "</ul></p>");
```

This code is used in §26.

**§37.** A convenient way to see if we've received requests of any given type:

```
int count_requests_of_type(int t) {
    request *req;
    int count = 0;
    LOOP_OVER(req, request)
        if (req->what_is_requested == t)
            count++;
    return count;
}
```

The function count_requests_of_type is.

# Solution Deviser 3/sol

*Purpose*

To make a solution (.sol) file accompanying a release, if requested.

*Definitions*

**¶1.** A solution file is simply a list of commands which will win a work of IF, starting from turn 1. In this section we will generate this list given the Skein file for an Inform 7 project: to follow this code, it's useful first to read the "Walkthrough solutions" section of the "Releasing" chapter in the main Inform documentation.

We will need to parse the entire skein into a tree structure, in which each node (including leaves) is one of the following structures. We expect the Inform user to have annotated certain nodes with the text `***` (three asterisks); the solution file will ignore all paths in the skein which do not lead to one of these `***` nodes. The surviving nodes, those in lines which do lead to `***` endings, are called "relevant".

Some knots have "branch descriptions", others do not. These are the options where choices have to be made. The `branch_parent` and `branch_count` fields are used to keep these labels: see below.

```
define MAX_NODE_ID_LENGTH 32
define MAX_COMMAND_LENGTH 128
define MAX_ANNOTATION_LENGTH 128
```

```
typedef struct skein_node {
    char id[MAX_NODE_ID_LENGTH];                uniquely identifying ID used within the Skein file
    char command[MAX_COMMAND_LENGTH];           text of the command at this node
    char annotation[MAX_ANNOTATION_LENGTH];     text of any annotation added by the user
    int relevant;                               is this node within one of the "relevant" lines in the skein?
    struct skein_node *branch_parent;           the trunk of the branch description, if any, is this way
    int branch_count;                           the leaf of the branch description, if any, is this number
    struct skein_node *parent;                  within the Skein tree: NULL for the root only
    struct skein_node *child;                   within the Skein tree: NULL if a leaf
    struct skein_node *sibling;                 within the Skein tree: NULL if the final option from its parent
    MEMORY_MANAGEMENT
} skein_node;
```

The structure skein_node is private to this section.

**¶2.** The root of the Skein, representing the start position before any command is typed, lives here:

```
skein_node *root_skn = NULL;                                only NULL when the tree is empty
```

---

**§1.** This section provides just one function to the rest of `cblorb`: this one, which implements the Blurb `solution` command.

Our method works in four steps. Steps 1 to 3 have a running time of $O(K^2)$, where $K$ is the number of knots in the Skein, and step 4 is $O(K \log_2(K))$, so the process as a whole is $O(K^2)$.

```
void walkthrough(char *Skein_filename, char *walkthrough_filename) {
    build_skein_tree(Skein_filename);
    if (root_skn == NULL) {
        error("there appear to be no threads in the Skein");
        return;
    }
    identify_relevant_lines();
    if (root_skn->relevant == FALSE) {
        error("no threads in the Skein have been marked '***'");
        return;
    }
    prune_irrelevant_lines();
    write_solution_file(walkthrough_filename);
}
```

The function walkthrough is called from 3/rel.

## §2. Step 1: building the Skein tree.

```
skein_node *current_skein_node = NULL;

void build_skein_tree(char *Skein_filename) {
    root_skn = NULL;
    current_skein_node = NULL;
    file_read(Skein_filename, "can't open skein file", FALSE, read_skein_pass_1, 0);
    current_skein_node = NULL;
    file_read(Skein_filename, "can't open skein file", FALSE, read_skein_pass_2, 0);
}

void read_skein_pass_1(char *line, text_file_position *tfp) { read_skein_line(line, 1); }
void read_skein_pass_2(char *line, text_file_position *tfp) { read_skein_line(line, 2); }
```

The function build_skein_tree is.

The function read_skein_pass_1 is.

The function read_skein_pass_2 is.

§**3.** The Skein is stored as an XML file. Its format was devised by Andrew Hunter in the early days of the Inform user interface for Mac OS X, and this was then adopted by the user interface on other platforms, so that projects could be freely exchanged between users regardless of their platforms. That makes it a kind of standard, but it isn't at present a public or documented one. We shall therefore make few assumptions about it.

```
void read_skein_line(char *line, int pass) {
    char node_id[MAX_NODE_ID_LENGTH];
    find_node_ID_in_tag(line, "item", node_id, MAX_NODE_ID_LENGTH, TRUE);
    if (pass == 1) {
        if (node_id[0]) ⟨Create a new skein tree node with this node ID 4⟩;
        if (current_skein_node) {
            ⟨Look for a "command" tag and set the command text from it 6⟩;
            ⟨Look for an "annotation" tag and set the annotation text from it 7⟩;
        }
    } else {
        if (node_id[0]) current_skein_node = find_node_with_ID(node_id);
        if (current_skein_node) {
            char child_node_id[MAX_NODE_ID_LENGTH];
            find_node_ID_in_tag(line, "child", child_node_id, MAX_NODE_ID_LENGTH, TRUE);
            if (child_node_id[0]) {
                skein_node *new_child = find_node_with_ID(child_node_id);
                if (new_child == NULL) {
                    error("the skein file is malformed (B)");
                    return;
                }
                ⟨Make the parent-child relationship 5⟩;
            }
        }
    }
}
```

The function read_skein_line is.

§**4.** Note that the root is the first knot in the Skein file.

```
⟨Create a new skein tree node with this node ID 4⟩ ≡
    current_skein_node = CREATE(skein_node);
    if (root_skn == NULL) root_skn = current_skein_node;
    strcpy(current_skein_node->id, node_id);
    strcpy(current_skein_node->command, "");
    strcpy(current_skein_node->annotation, "");
    current_skein_node->branch_count = -1;
    current_skein_node->branch_parent = NULL;
    current_skein_node->parent = NULL;
    current_skein_node->child = NULL;
    current_skein_node->sibling = NULL;
    current_skein_node->relevant = FALSE;
    if (trace_mode) printf("Creating knot with ID '%s'\n", node_id);
```

This code is used in §3.

**§5.** We make `new_child` the youngest child of `current_skein_mode`:

⟨Make the parent-child relationship 5⟩ ≡

```
    new_child->parent = current_skein_node;
    if (current_skein_node->child == NULL) {
        current_skein_node->child = new_child;
    } else {
        skein_node *familial = current_skein_node->child;
        while (familial->sibling) familial = familial->sibling;
        familial->sibling = new_child;
    }
```

This code is used in §3.

**§6.**

⟨Look for a “command” tag and set the command text from it 6⟩ ≡

```
    char *p = current_skein_node->command;
    if (find_text_of_tag(line, "command", p, MAX_COMMAND_LENGTH, FALSE)) {
        if (trace_mode) printf("Raw command '%s'\n", p);
        undo_XML_escapes_in_string(p);
        convert_string_to_upper_case(p);
        if (trace_mode) printf("Processed command '%s'\n", p);
    }
```

This code is used in §3.

**§7.**

⟨Look for an “annotation” tag and set the annotation text from it 7⟩ ≡

```
    char *p = current_skein_node->annotation;
    if (find_text_of_tag(line, "annotation", p, MAX_ANNOTATION_LENGTH, FALSE)) {
        if (trace_mode) printf("Raw annotation '%s'\n", p);
        undo_XML_escapes_in_string(p);
        if (trace_mode) printf("Processed annotation '%s'\n", p);
    }
```

This code is used in §3.

**§8.** Try to find a node ID element attached to a particular tag on the line:

```
int find_node_ID_in_tag(char *line, char *tag,
    char *write_to, int max_length, int abort_not_trim) {
    char portion1[MAX_TEXT_FILE_LINE_LENGTH], portion2[MAX_TEXT_FILE_LINE_LENGTH];
    char prototype[64];
    strcpy(prototype, "%[^<]<");
    strcat(prototype, tag);
    strcat(prototype, " nodeId=\"%[^\"]\"");
    write_to[0] = 0;
    if (sscanf(line, prototype, portion1, portion2) == 2) {
        if ((strlen(portion2) >= max_length-1) && (abort_not_trim)) {
            error("the skein file is malformed (C)");
            return FALSE;
        }
        strncpy(write_to, portion2, max_length-1); write_to[max_length-1] = 0;
        return TRUE;
    }
    return FALSE;
}
```

The function find_node_ID_in_tag is.

**§9.** Try to find the text of a particular tag on the line:

```
int find_text_of_tag(char *line, char *tag,
    char *write_to, int max_length, int abort_not_trim) {
    char portion1[MAX_TEXT_FILE_LINE_LENGTH], portion2[MAX_TEXT_FILE_LINE_LENGTH],
        portion3[MAX_TEXT_FILE_LINE_LENGTH];
    char prototype[64];
    strcpy(prototype, "%[^>]>%[^<]</");
    strcat(prototype, tag);
    strcat(prototype, "%s");
    if (sscanf(line, prototype, portion1, portion2, portion3) == 3) {
        if ((strlen(portion2) >= max_length-1) && (abort_not_trim)) {
            error("the skein file is malformed (C)");
            return FALSE;
        }
        strncpy(write_to, portion2, max_length-1); write_to[max_length-1] = 0;
        if (trace_mode) printf("found %s = '%s'\n", tag, portion2);
        return TRUE;
    }
    return FALSE;
}
```

The function find_text_of_tag is.

**§10.** This is not very efficient, but:

```
skein_node *find_node_with_ID(char *id) {
    skein_node *skn;
    LOOP_OVER(skn, skein_node)
        if (strcmp(id, skn->id) == 0)
            return skn;
    return NULL;
}
```

The function find_node_with_ID is.

**§11.** Finally, we needed the following string hackery:

```
void convert_string_to_upper_case(char *p) {
    int i;
    for (i=0; p[i]; i++) p[i]=toupper(p[i]);
}
```

The function convert_string_to_upper_case is.

**§12.** and:

```
void undo_XML_escapes_in_string(char *p) {
    int i = 0, j = 0;
    while (p[i]) {
        if (p[i] == '&') {
            char xml_escape[16];
            int k=0;
            while ((p[i+k] != 0) && (p[i+k] != ';') && (k<14)) {
                xml_escape[k] = tolower(p[i+k]); k++;
            }
            xml_escape[k] = p[i+k]; k++; xml_escape[k] = 0;
            ⟨We have identified an XML escape 13⟩;
        }
        p[j++] = p[i++];
    }
    p[j++] = 0;
}
```

The function undo_XML_escapes_in_string is.

**§13.** Note that all other ampersand-escapes are passed through verbatim.

```
⟨We have identified an XML escape 13⟩ ≡
    char c = 0;
    if (strcmp(xml_escape, "&lt;") == 0) c = '<';
    if (strcmp(xml_escape, "&gt;") == 0) c = '>';
    if (strcmp(xml_escape, "&amp;") == 0) c = '&';
    if (strcmp(xml_escape, "&apos;") == 0) c = '\'';
    if (strcmp(xml_escape, "&quot;") == 0) c = '\"';
    if (c) { p[j++] = c; i += strlen(xml_escape); continue; }
```

This code is used in §12.

**§14. Step 2: identify the relevant lines.** We aim to show how to reach all knots in the Skein annotated with text which begins with three asterisks. (We trim those asterisks away from the annotation once we spot them: they have served their purpose.) A knot is "relevant" if and only if one of its (direct or indirect) children is marked with three asterisks in this way.

```
void identify_relevant_lines(void) {
    skein_node *skn;
    LOOP_OVER(skn, skein_node) {
        char *p = skn->annotation;
        if (trace_mode) printf("Knot %s is annotated '%s'\n", skn->id, p);
        if ((p[0] == '*') && (p[1] == '*') && (p[2] == '*')) {
            int i = 3, j; while (p[i] == ' ') i++;
            for (j=0; p[i]; i++) p[j++] = p[i]; p[j] = 0;
            skein_node *knot;
            for (knot = skn; knot; knot = knot->parent) {
                knot->relevant = TRUE;
                if (trace_mode) printf("Knot %s is relevant\n", knot->id);
            }
        }
    }
}
```

The function identify_relevant_lines is.

**§15. Step 3: pruning irrelevant lines out of the tree.** When the loop below concludes, the relevant nodes are exactly those in the component of the tree root, because:

(a) No irrelevant node can be the child of a relevant one; and no relevant node can be the child of an irrelevant one by definition. So the tree falls into components each of which is fully relevant or fully not.
(b) Since we never break any relevant-parent-relevant-child relationships, the number of components containing at least one relevant node is unchanged.
(c) Since the Skein is initially a tree and not a forest, we start with just one component, and it contains the tree root, which is known to be relevant (we would have given up with an error message if not).
(d) And therefore at the end of the loop the "tree" consists of a single component headed by the tree root and containing all of the relevant nodes, together with any number of other components each of which contains only irrelevant ones.

```
void prune_irrelevant_lines(void) {
    skein_node *skn;
    LOOP_OVER(skn, skein_node)
        if ((skn->relevant == FALSE) && (skn->parent))
            ⟨Delete this node from its parent 16⟩;
}
```

The function prune_irrelevant_lines is.

**§16.**

⟨Delete this node from its parent 16⟩ ≡

```
    if (skn->parent->child == skn) {
        skn->parent->child = skn->sibling;
    } else {
        skein_node *skn2 = skn->parent->child;
        while ((skn2) && (skn2->sibling != skn)) skn2 = skn2->sibling;
        if ((skn2) && (skn2->sibling == skn)) skn2->sibling = skn->sibling;
    }
    skn->parent = NULL;
    skn->sibling = NULL;
```

This code is used in §15.

**§17. Step 4: writing the solution file.**

```
void write_solution_file(char *walkthrough_filename) {
    FILE *SOL = fopen(walkthrough_filename, "w");
    if (SOL == NULL)
        fatal_fs("unable to open destination for solution text file",
            walkthrough_filename);
    fprintf(SOL, "Solution to \""); copy_placeholder_to("TITLE", SOL);
    fprintf(SOL, "\" by "); copy_placeholder_to("AUTHOR", SOL); fprintf(SOL, "\n\n");
    recursively_solve(SOL, root_skn, NULL);
    fclose(SOL);
}
```

The function write_solution_file is.

**§18.** The following prints commands to the solution file from the position `skn` – which means just after typing its command – with the aim of reaching all relevant endings we can get to from there.

```
void recursively_solve(FILE *SOL, skein_node *skn, skein_node *last_branch) {
    ⟨Follow the skein down until we reach a divergence, if we do 19⟩;
    ⟨Print the various alternatives from this knot where the threads diverge 20⟩;
    ⟨Show the solutions down each of these alternative lines in turn 21⟩;
}
```

The function recursively_solve is.

**§19.** If there's only a single option from here, we could print it and then call `recursively_solve` down from it. That would make the code shorter and clearer, perhaps, but it would clobber the C stack: our recursion depth might be into the tens of thousands on long solution files. So we tail-recurse instead of calling ourselves, so to speak, and just run down the thread until we reach a choice. (If we never do reach a choice, we can return – there is nowhere else to reach.)

⟨Follow the skein down until we reach a divergence, if we do 19⟩ ≡

```
    while ((skn->child == NULL) || (skn->child->sibling == NULL)) {
        if (skn->child == NULL) return;
        if (skn->child->sibling == NULL) {
            skn = skn->child;
            write_command(SOL, skn, NORMAL_COMMAND);
        }
    }
```

This code is used in §18.

**§20.** Thus we are here only when there are at least two alternative commands we might use from position skn.

⟨Print the various alternatives from this knot where the threads diverge 20⟩ ≡

```
    fprintf(SOL, "Choice:\n");
    int branch_counter = 1;
    skein_node *option;
    for (option = skn->child; option; option = option->sibling)
        if (option->child == NULL) {
            write_command(SOL, option, BRANCH_TO_END_COMMAND);
        } else {
            option->branch_count = branch_counter++;
            option->branch_parent = last_branch;
            write_command(SOL, option, BRANCH_TO_LINE_COMMAND);
        }
```

This code is used in §18.

**§21.**

⟨Show the solutions down each of these alternative lines in turn 21⟩ ≡

```
    skein_node *option;
    for (option = skn->child; option; option = option->sibling)
        if (option->child) {
            fprintf(SOL, "\nBranch (");
            write_branch_name(SOL, option);
            fprintf(SOL, ")\n");
            recursively_solve(SOL, option, option);
        }
```

This code is used in §18.

## §22. Writing individual commands and branch descriptions.

**define** NORMAL_COMMAND 1
**define** BRANCH_TO_END_COMMAND 2
**define** BRANCH_TO_LINE_COMMAND 3

```
void write_command(FILE *SOL, skein_node *cmd_skn, int form) {
    if (form != NORMAL_COMMAND) fprintf(SOL, "  ");
    fprintf(SOL, "%s", cmd_skn->command);
    if (form != NORMAL_COMMAND) {
        fprintf(SOL, " -> ");
        if (form == BRANCH_TO_LINE_COMMAND) {
            fprintf(SOL, "go to branch (");
            write_branch_name(SOL, cmd_skn);
            fprintf(SOL, ")");
        }
        else fprintf(SOL, "end");
    }
    if (cmd_skn->annotation[0]) fprintf(SOL, " ... %s", cmd_skn->annotation);
    fprintf(SOL, "\n");
}
```

The function write_command is.

**§23.** For instance, at the third option from a thread which ran back to being the second option from a thread which ran back to being the seventh option from the original position, the following would print "7.2.3". Note that only the knots representing the positions after commands which make a choice are labelled in this way.

```
void write_branch_name(FILE *SOL, skein_node *skn) {
    if (skn->branch_parent) {
        write_branch_name(SOL, skn->branch_parent);
        fprintf(SOL, ".");
    }
    fprintf(SOL, "%d", skn->branch_count);
}
```

The function write_branch_name is.

# Links and Auxiliary Files 3/links

*Purpose*

To manage links to auxiliary files, and placeholder variables.

*Definitions*

**¶1.** Auxiliary files are for items bundled up with the release but which are deliberately made accessible for the eventual player: things such as maps or manuals. `cblorb` needs to know about these only when releasing a website; they are also recorded in an iFiction record, but `cblorb` does not create that (`ni` does).

```
typedef struct auxiliary_file {
    char relative_URL[MAX_FILENAME_LENGTH];
    char full_filename[MAX_FILENAME_LENGTH];
    char aux_leafname[MAX_FILENAME_LENGTH];
    char description[MAX_FILENAME_LENGTH];
    char format[MAX_EXTENSION_LENGTH];                          e.g., "jpg", "pdf"
    MEMORY_MANAGEMENT
} auxiliary_file;
```

The structure auxiliary_file is private to this section.

**§1. Registration.** The format text is set to a lower-case version of the filename extension, and the URL to the filename itself; except when there is no extension, so that the auxiliary resource is a mini-website in a subfolder of the release website. In that case the format is `link` and the URL is to the index file in the subfolder.

```
void create_auxiliary_file(char *filename, char *description) {
    auxiliary_file *aux = CREATE(auxiliary_file);
    strcpy(aux->description, description);
    strcpy(aux->full_filename, filename);
    char *ext = get_filename_extension(filename);
    char *leaf = get_filename_leafname(filename);
    if (ext[0] == '.') {
        strcpy(aux->relative_URL, filename);
        if (strlen(ext + 1) >= MAX_EXTENSION_LENGTH - 1) {
            error("auxiliary file has overlong extension"); return;
        }
        strcpy(aux->format, ext + 1);
        int k; for (k=0; aux->format[k]; k++) aux->format[k] = tolower(aux->format[k]);
    } else {
        strcpy(aux->format, "link");
        sprintf(aux->relative_URL, "%s%cindex.html", filename, SEP_CHAR);
    }
    strcpy(aux->aux_leafname, leaf);
    printf("! Auxiliary file: <%s> = <%s>\n", filename, description);
}
```

The function create_auxiliary_file is called from 1/blurb.

§**2. Linking.** The list of links to auxiliary resources is written using `<li>...</li>` list entry tags, for convenience of CSS styling.

```
void expand_AUXILIARY_variable(FILE *COPYTO) {
    auxiliary_file *aux;
    LOOP_OVER(aux, auxiliary_file) {
        fprintf(COPYTO, "<li>");
        download_link(COPYTO,
            aux->description, aux->full_filename, aux->aux_leafname, aux->format);
        fprintf(COPYTO, "</li>");
    }
    add_links_to_requested_resources(COPYTO);
}
```

The function expand_AUXILIARY_variable is.

§**3.** On some of the pages produced by `cblorb` the story file itself looks like another auxiliary resource, but it's produced thus:

```
void expand_DOWNLOAD_variable(FILE *COPYTO) {
    char target_pathname[MAX_FILENAME_LENGTH];          eventual pathname of Blorb file written
    sprintf(target_pathname, "%s%c%s", release_folder, SEP_CHAR, read_placeholder("STORYFILE"));
    download_link(COPYTO, "Story File", target_pathname, read_placeholder("STORYFILE"), "Blorb");
}
```

The function expand_DOWNLOAD_variable is.

§**4. Links.** This routine, then, handles either kind of link.

```
void download_link(FILE *COPYTO, char *desc, char *filename, char *relative_url, char *form) {
    int size_up = TRUE;
    if (strcmp(form, "link") == 0) size_up = FALSE;
    fprintf(COPYTO, "<a href=\"%s\">%s</a> ", relative_url, desc);
    open_style(COPYTO, "filetype");
    fprintf(COPYTO, "(%s", form);
    if (size_up) {
        long int size = -1L;
        if (strcmp(desc, "Story File") == 0) size = (long int) blorb_file_size;
        else size = file_size(filename);
        if (size != -1L) ⟨Write a description of the rough file size 5⟩
    }
    fprintf(COPYTO, ")");
    close_style(COPYTO, "filetype");
}
```

The function download_link is called from 3/rel.

**§5.** We round down to the nearest KB, MB, GB, TB or byte, as appropriate. Although this will describe a 1-byte auxiliary file as "1 bytes", the contingency seems remote.

⟨Write a description of the rough file size 5⟩ ≡

```
    char *units = "&nbsp;bytes";
    long int remainder = 0;
    if (size > 1024L) { remainder = size % 1024L; size /= 1024L; units = "KB"; }
    if (size > 1024L) { remainder = size % 1024L; size /= 1024L; units = "MB"; }
    if (size > 1024L) { remainder = size % 1024L; size /= 1024L; units = "GB"; }
    if (size > 1024L) { remainder = size % 1024L; size /= 1024L; units = "TB"; }
    fprintf(COPYTO, ",&nbsp;%d", (int) size);
    if ((size < 100L) && (remainder >= 103L)) fprintf(COPYTO, ".%d", (int) (remainder/103L));
    fprintf(COPYTO, "%s", units);
```

This code is used in §4.

**§6. Cover image.** Note that if the large cover image is a PNG, so is the small (thumbnail) version, and vice versa – supplying "Cover.jpg" and "Small Cover.png" will not work.

```
void expand_COVER_variable(FILE *COPYTO) {
    if (cover_exists) {
        char *format = "png"; if (cover_is_in_JPEG_format) format = "jpg";
        fprintf(COPYTO, "<a href=\"Cover.%s\"><img src=\"Small Cover.%s\" border=\"1\" /></a>",
            format, format);
    }
}
```

The function expand_COVER_variable is.

**§7. Releasing.** When we generate a website, we need to copy the auxiliary files into it (though not mini-websites: the user will have to do that).

```
void request_copy_of_auxiliaries(void) {
    auxiliary_file *aux;
    LOOP_OVER(aux, auxiliary_file)
        if (strcmp(aux->format, "link") != 0) {
            if (trace_mode)
                printf("! COPY <%s> as <%s>\n", aux->full_filename, aux->aux_leafname);
            request_copy(aux->full_filename, aux->aux_leafname);
        }
}
```

The function request_copy_of_auxiliaries is called from 3/rel.

# Placeholders 3/place

*Purpose*

To manage placeholder variables.

*Definitions*

**¶1.** Placeholders are markers such as "[AUTHOR]", found in the template files for making web pages. ("AUTHOR" would be the name of this one; the use of capital letters is customary but not required.) Most of these can be set to arbitrary texts by use of the `placeholder` command in the blurb file, but a few are "reserved" by `cblorb`:

```
define SOURCE_RPL 1
define SOURCENOTES_RPL 2
define SOURCELINKS_RPL 3
define COVER_RPL 4
define DOWNLOAD_RPL 5
define AUXILIARY_RPL 6
define PAGENUMBER_RPL 7
define PAGEEXTENT_RPL 8
```

```
typedef struct placeholder {
    char pl_name[MAX_VAR_NAME_LENGTH];
    char pl_contents[MAX_FILENAME_LENGTH];                     current value
    int reservation;                  one of the *_RPL values above, or 0 for unreserved
    int locked;               currently being expanded: locked to prevent mise-en-abyme
    MEMORY_MANAGEMENT
} placeholder;
```

The structure placeholder is private to this section.

**§1. Initial values.** The `BLURB` refers here to back-cover-style text, and not to the "blurb" file which we are acting on.

```
void initialise_placeholders(void) {
    set_placeholder_to("SOURCE", "", SOURCE_RPL);
    set_placeholder_to("SOURCENOTES", "", SOURCENOTES_RPL);
    set_placeholder_to("SOURCELINKS", "", SOURCELINKS_RPL);
    set_placeholder_to("COVER", "", COVER_RPL);
    set_placeholder_to("DOWNLOAD", "", DOWNLOAD_RPL);
    set_placeholder_to("AUXILIARY", "", AUXILIARY_RPL);
    set_placeholder_to("PAGENUMBER", "", PAGENUMBER_RPL);
    set_placeholder_to("PAGEEXTENT", "", PAGEEXTENT_RPL);
    set_placeholder_to("CBLORBERRORS", "", 0);
    set_placeholder_to("INBROWSERPLAY", "", 0);
    set_placeholder_to("BLURB", "", 0);
    set_placeholder_to("TEMPLATE", "Standard", 0);
    set_placeholder_to("GENERATOR", VERSION, 0);
    set_placeholder_to("BASE64_TOP", "", 0);
    set_placeholder_to("BASE64_TAIL", "", 0);
    set_placeholder_to("JAVASCRIPTPRELUDE", JAVASCRIPT_PRELUDE, 0);
    set_placeholder_to("FONTTAG", FONT_TAG, 0);
    initialise_time_variables();
}
```

The function initialise_placeholders is called from 1/main.

§**2.** We don't need any very efficient system for parsing these names, as there are typically fewer than 20 placeholders at a time.

```
placeholder *find_placeholder(char *name) {
    placeholder *wv;
    LOOP_OVER(wv, placeholder)
        if (strcmp(wv->pl_name, name) == 0)
            return wv;
    return NULL;
}
char *read_placeholder(char *name) {
    placeholder *wv = find_placeholder(name);
    if (wv) return wv->pl_contents;
    return NULL;
}
```

The function find_placeholder is.
The function read_placeholder is called from 1/main, 1/blurb, 3/rel, 3/links and 3/web.

§**3.** There are no "types" of these placeholders. When they hold numbers, it's only as the text of a number written out in decimal, so:

```
void set_placeholder_to_number(char *var, int v) {
    char temp_digits[64];
    sprintf(temp_digits, "%d", v);
    set_placeholder_to(var, temp_digits, 0);
}
```

The function set_placeholder_to_number is called from 1/main and 1/blurb.

§**4.** And here we set a given placeholder to a given text value. If it doesn't already exist, it will be created. A reserved placeholder can then never again be set, and since it will have been set at creation time (above), it follows that a reserved placeholder cannot be set with the `placeholder` command of a blurb file.

```
void set_placeholder_to(char *var, char *text, int reservation) {
    set_placeholder_to_inner(var, text, reservation, FALSE);
}
void append_to_placeholder(char *var, char *text) {
    set_placeholder_to_inner(var, text, 0, TRUE);
}
```

The function set_placeholder_to is called from 1/main, 1/blurb and 3/rel.
The function append_to_placeholder is called from 1/text, 1/blurb and 3/rel.

**§5.** Where:

```
void set_placeholder_to_inner(char *var, char *text, int reservation, int extend) {
    if (strlen(var) >= MAX_VAR_NAME_LENGTH-1) { error("variable name too long"); return; }

    if (trace_mode) printf("! [%s] <-- \"%s\"\n", var, (text)?text:"");

    placeholder *wv = find_placeholder(var);
    if ((wv) && (reservation > 0)) { error("tried to set reserved variable"); return; }
    if (wv == NULL) {
        wv = CREATE(placeholder);
        if (trace_mode) printf("! Creating [%s]\n", var);
        strcpy(wv->pl_name, var);
        (wv->pl_contents)[0] = 0;
        wv->reservation = reservation;
    }

    int L = strlen(text) + 1;
    if (extend) L += strlen(wv->pl_contents);
    if (L >= MAX_FILENAME_LENGTH) { error("placeholder text too long"); return; }

    if (extend) strcat(wv->pl_contents, text);
    else strcpy(wv->pl_contents, text);
}
```

The function set_placeholder_to_inner is.

**§6.** And that just leaves writing the output of these placeholders. The scenario here is that we're copying HTML over to make a new web page, but we've hit text in the template like "[AUTHOR]". We output the value of this placeholder instead of that literal text. The reserved placeholders output as special gadgets instead of any fixed text, so those all call suitable routines elsewhere in `cblorb`.

If the placeholder name isn't known to us, we print the text back, so that the original material will be unchanged. (This is in case the original contains uses of square brackets which aren't for placeholding.)

```
int escape_quotes_mode = 0;
void copy_placeholder_to(char *var, FILE *COPYTO) {
    int multiparagraph_mode = FALSE, eqm = escape_quotes_mode;
    if (var[0] == '*') { var++; escape_quotes_mode = 1; }
    if (var[0] == '*') { var++; escape_quotes_mode = 2; }
    if (strcmp(var, "BLURB") == 0) multiparagraph_mode = TRUE;
    placeholder *wv = find_placeholder(var);
    if ((wv == NULL) || (wv->locked)) {
        fprintf(COPYTO, "[%s]", var);
    } else {
        wv->locked = TRUE;
        if (multiparagraph_mode) fprintf(COPYTO, "<p>");
        switch (wv->reservation) {
            case 0: ⟨Copy an ordinary unreserved placeholder 7⟩; break;
            case SOURCE_RPL: expand_SOURCE_or_SOURCENOTES_variable(COPYTO, FALSE); break;
            case SOURCENOTES_RPL: expand_SOURCE_or_SOURCENOTES_variable(COPYTO, TRUE); break;
            case SOURCELINKS_RPL: expand_SOURCELINKS_variable(COPYTO); break;
            case COVER_RPL: expand_COVER_variable(COPYTO); break;
            case DOWNLOAD_RPL: expand_DOWNLOAD_variable(COPYTO); break;
            case AUXILIARY_RPL: expand_AUXILIARY_variable(COPYTO); break;
            case PAGENUMBER_RPL: expand_PAGENUMBER_variable(COPYTO); break;
            case PAGEEXTENT_RPL: expand_PAGEEXTENT_variable(COPYTO); break;
        }
        if (multiparagraph_mode) fprintf(COPYTO, "</p>");
        wv->locked = FALSE;
        escape_quotes_mode = eqm;
    }
}
```

The function copy_placeholder_to is called from 3/sol and 3/web.

**§7.** Note that the [BLURB] placeholder – which holds the story description, and is like a back cover blurb for a book; the name is not related to the release instructions format – may consist of multiple paragraphs. If so, then they will be divided by `<br/>`, since that's the XML convention. But we want to translate those breaks to `</p><p>`, closing an old paragraph and opening a new one, because that will make the blurb text much easier to style with a CSS file. It follows that [BLURB] should always appear in templates within an HTML paragraph.

⟨Copy an ordinary unreserved placeholder 7⟩ ≡

```
    int i; char *p = wv->pl_contents;
    for (i=0; p[i]; i++) {
        if ((p[i] == '<') && (p[i+1] == 'b') && (p[i+2] == 'r') &&
            (p[i+3] == '/') && (p[i+4] == '>') && (multiparagraph_mode)) {
            fprintf(COPYTO, "</p><p>"); i += 4; continue;
        }
        if (p[i] == '[') {
            char inner_name[MAX_VAR_NAME_LENGTH+1];
            int j = i+1, k = 0, expanded = FALSE; inner_name[0] = 0;
            for (; p[j]; j++) {
                if ((p[j] == '[') || (p[j] == ' ')) break;
                if (p[j] == ']') {
                    i = j;
                    copy_placeholder_to(inner_name, COPYTO);
                    expanded = TRUE;
                    break;
                }
                inner_name[k++] = p[j]; inner_name[k] = 0;
                if (k >= MAX_VAR_NAME_LENGTH) break;
            }
            if (expanded) continue;
        }
        if (((p[i] == '\x0a') || (p[i] == '\x0d') || (p[i] == '\x7f')) &&
            (multiparagraph_mode)) {
            fprintf(COPYTO, "<p>"); continue;
        }
        if ((escape_quotes_mode == 1) && (p[i] == '\'')) fprintf(COPYTO, "&#39;");
        else if ((escape_quotes_mode == 2) && (p[i] == '\'')) fprintf(COPYTO, "%%2527");
        else fprintf(COPYTO, "%c", p[i]);
    }
```

This code is used in §6.

# Templates 3/templ

*Purpose*

To manage templates for website generation.

*Definitions*

**¶1.** Template paths define, in order of priority, where to look for templates.

```
typedef struct template_path {
    char template_repository[MAX_FILENAME_LENGTH];          pathname of folder of repository
    MEMORY_MANAGEMENT
} template_path;
```

The structure template_path is private to this section.

**¶2.** Templates are the things themselves.

```
typedef struct template {
    char template_name[MAX_FILENAME_LENGTH];                 e.g., "Standard"
    struct template_path *template_location;
    char latest_use[MAX_FILENAME_LENGTH];                    filename most recently sought from it
    MEMORY_MANAGEMENT
} template;
```

The structure template is private to this section.

## §1. Defining template paths.

The following implements the Blurb command "template path".

```
int no_template_paths = 0;
void new_template_path(char *pathname) {
    template_path *tp = CREATE(template_path);
    strcpy(tp->template_repository, pathname);
    if (trace_mode)
        printf("! Template search path %d: <%s>\n", ++no_template_paths, pathname);
}
```

The function new_template_path is called from 1/blurb.

**§2.** The following searches for a named file in a named template, returning the template path which holds the template if it exists. This might look a pretty odd thing to do – weren't we looking the file itself? But the answer is that `seek_file_in_template_paths` is really used to detect the presence of templates, not of files.

```
template_path *seek_file_in_template_paths(char *name, char *leafname) {
    template_path *tp;
    LOOP_OVER(tp, template_path) {
        char possible[MAX_FILENAME_LENGTH];
        sprintf(possible, "%s%c%s%c%s",
            tp->template_repository, SEP_CHAR, name, SEP_CHAR, leafname);
        if (file_exists(possible)) return tp;
    }
    return NULL;
}
```

The function seek_file_in_template_paths is.

**§3.** And this is where that happens. Suppose we need to locate the template "Molybdenum". We ought to do this by looking for a directory of that name among the template paths, but searching for directories is a little tricky to do in ANSI C in a way which will work on all platforms. So instead we look for any of the four files which compulsorily ought to exist (or the one which does in the case of an interpreter; those look rather like website templates).

```
template *find_template(char *name) {
    template *t;
    ⟨Is this a template we already know? 4⟩;
    template_path *tp = seek_file_in_template_paths(name, "index.html");
    if (tp == NULL) tp = seek_file_in_template_paths(name, "source.html");
    if (tp == NULL) tp = seek_file_in_template_paths(name, "style.css");
    if (tp == NULL) tp = seek_file_in_template_paths(name, "(extras).txt");
    if (tp == NULL) tp = seek_file_in_template_paths(name, "(manifest).txt");
    if (tp) {
        t = CREATE(template);
        strcpy(t->template_name, name);
        t->template_location = tp;
        return t;
    }
    return NULL;
}
```

The function find_template is.

**§4.** It reduces pointless file accesses to cache the results, so:

```
⟨Is this a template we already know? 4⟩ ≡
    LOOP_OVER(t, template)
        if (strcmp(name, t->template_name) == 0)
            return t;
```

This code is used in §3.

**§5. Searching for template files.** If we can't find the file `name` in the template specified, we try looking inside "Standard" instead (if we can find a template of that name).

```
int template_doesnt_exist = FALSE;
char *find_file_in_named_template(char *name, char *needed) {
    template *t = find_template(name), *Standard = find_template("Standard");
    if (t == NULL) {
        if (template_doesnt_exist == FALSE) {
            errorf_1s(
                "Websites and play-in-browser interpreter web pages are created "
                "using named templates. (Basic examples are built into the Inform "
                "application. You can also create your own, putting them in the "
                "'Templates' subfolder of the project's Materials folder.) Each "
                "template has a name. On this Release, I tried to use the "
                "'%s' template, but couldn't find a copy of it anywhere.", name);
        }
        template_doesnt_exist = TRUE;
    }
    char *path = try_single_template(t, needed);
    if ((path == NULL) && (Standard))
        path = try_single_template(Standard, needed);
    return path;
}
```

The function find_file_in_named_template is called from 3/rel.

**§6.** Where, finally:

```
char *try_single_template(template *t, char *needed) {
    if (t == NULL) return NULL;
    sprintf(t->latest_use, "%s%c%s%c%s",
        t->template_location->template_repository, SEP_CHAR, t->template_name, SEP_CHAR, needed);
    if (trace_mode) printf("! Trying <%s>\n", t->latest_use);
    if (file_exists(t->latest_use)) return t->latest_use;
    return NULL;
}
```

The function try_single_template is.

# Website Maker 3/web

*Purpose*

To accompany a release with a mini-website.

*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, "&nbsp;&nbsp;&nbsp;&nbsp;");
    }
}

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, "&nbsp;&nbsp;</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 §39,47,39.

**§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 §39,47,39.

§**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.

# Base64 3/b64

*Purpose*

To produce base64-encoded story files ready for in-browser play by a Javascript-based interpreter such as Parchment.

§**1. Base 64.** This encoding scheme is defined by the Internet standard RFC 1113. Broadly, the idea is to take a binary stream of bytes, break it into threes, and then convert this into a sequence of four emailable characters. To encode 24 bits in four characters, we need six bits per character, so we need $2^6 = 64$ characters in all. Since $64 = 26 + 26 + 10 + 2$, we can nearly get there with alphanumeric characters alone, adding just two others – conventionally, plus and forward-slash. That's more or less the whole thing, except that we use an equals sign to indicate incompleteness of the final triplet (which might have only 1 or 2 bytes in it).

RFC 1113 permits white space to be used freely, including in particular line breaks, but we don't avail ourselves.

```
char *RFC1113_table = "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789+/=";
```

§**2.** The encoding routine is as follows.

```
void encode_as_base64(char *in_filename, char *out_filename, char *top, char *tail) {
    FILE *IN = fopen(in_filename, "rb");
    if (IN == NULL)
        fatal_fs("can't open story file for base-64 encoding", in_filename);
    FILE *OUT = fopen(out_filename, "w");                    a text file, not binary
    if (OUT == NULL)
        fatal_fs("can't open base-64 encoded story file for output", out_filename);
    if (top) fprintf(OUT, "%s", top);
    while (TRUE) {
        int triplet[3], triplet_size = 0;
        ⟨Read the triplet of binary bytes, storing 0 to 3 in the size read 3⟩;
        if (triplet_size == 0) break;
        int quartet[4];
        ⟨Convert triplet to a quartet 4⟩;
        int i; for (i=0; i<4; i++) fputc(RFC1113_table[quartet[i]], OUT);
        if (triplet_size < 3) break;
    }
    if (tail) fprintf(OUT, "%s", tail);
    fclose(IN); fclose(OUT);
}
```

The function encode_as_base64 is called from 3/rel.

**§3.** If the file ends in mid-triplet, we pad out with zeros.

⟨Read the triplet of binary bytes, storing 0 to 3 in the size read 3⟩ ≡

```
triplet[0] = fgetc(IN);
if (triplet[0] != EOF) {
    triplet_size++;
    triplet[1] = fgetc(IN);
    if (triplet[1] != EOF) {
        triplet_size++;
        triplet[2] = fgetc(IN);
        if (triplet[2] != EOF)
            triplet_size++;
    }
}
int i; for (i=triplet_size; i<3; i++) triplet[i] = 0;
```

This code is used in §2.

**§4.**

⟨Convert triplet to a quartet 4⟩ ≡

```
int i; for (i=0; i<4; i++) quartet[i] = 0;
quartet[0] += (triplet[0] & 0xFC) >> 2;
quartet[1] += (triplet[0] & 0x03) << 4;
quartet[1] += (triplet[1] & 0xF0) >> 4;
quartet[2] += (triplet[1] & 0x0F) << 2;
quartet[2] += (triplet[2] & 0xC0) >> 6;
quartet[3] += (triplet[2] & 0x3F) << 0;
switch (triplet_size) {
    case 1: quartet[2] = 64; quartet[3] = 64; break;
    case 2: quartet[3] = 64; break;
}
```

This code is used in §2.