CBLORB

The Program

Complete Program

Build 2/090104 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.*

The cblorb Manual P/man

Purpose

A guide for users of cblorb.

---

§**1. Some definitions.** `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.

§**2.** "Blorb" is a general-purpose wrapper format designed as a way to gather together audiovisual media and bibliographic data for works of IF. The format was devised and formally specified by Andrew Plotkin around 2000, and its name is borrowed from that of a magic spell in Infocom's classic work, *Enchanter*. ("The blorb spell (safely protect a small object as though in a strong box).") Although Inform 6, the then current version, did not itself generate blorb files, a Perl script called `perlblorb` was provided in 2001 so that the user could perform the wrapping-up process. `perlblorb` is no longer used, and survives only in the name of `cblorb`, which is a C version of what had previously been written in Perl. This means it can run on machines with no Perl installation, which Inform 7 needs to be able to do. Unlike `perlblorb`, `cblorb` is "under the hood"; the user does not need to give it instructions. This manual is therefore useful only for people needing to generate Inform-related websites, or who are maintaining the Inform user interface applications.

§**3.** Sentences in Inform source text such as:

> Release along with public source text, cover art, and a website.

do in effect transmit instructions to `cblorb`, but `cblorb` doesn't read them in this natural-language form. Instead, the `ni` component of Inform 7 translates these instructions into a script for `cblorb` to follow. This script is called a "blurb".

"Blurb" is a mini-language for specifying how the materials in a work of IF should be packaged up for release. It was originally codified in 2001 as a standard way to describe how a blorb file should be put together, but it was extended in 2005 and again in 2008 so that it could also organise accompanying files released along with the blorb.

The original Blurb language was documented in chapter 43 of the DM4 (i.e., the *Inform Designer's Manual*, fourth edition, 2001); for clarity, we will call that language "Blurb 2001". Today's Blurb language is a little different. Some features of Blurb 2001 are deprecated and no longer used, while numerous other syntaxes are new. Because of this the DM4 specification is no longer useful, so we will give a full description below of Blurb as it currently stands.

§**4.** `cblorb` **within the Inform user interface.** This is the sequence of events when the user clicks Release in the user interface application (the "interface"):

(1) The interface calls `ni`, the I7 compiler, as normal except that the `-release` command-line switch is specified.

(2) `ni` compiles the source text into I6 code. If Problems occur, `ni` exits with a return code of 1, and the interface displays those, and then stops the process.

(3) If no Problems occur, `ni` writes two additional files besides the I6 code it always writes:
   (a) `Metadata.iFiction`, an iFiction record;
   (b) `Release.blurb`, a blurb file of instructions for `cblorb` to follow later.

(4) `ni` having returned 0 to indicate success, the interface next calls the Inform 6 compiler (called, e.g., `inform-6.31-biplatform`, but we'll call it `i6` here). The interface calls `i6` as normal except that the `S` and `D` switches, for strict checking and for debugging, are off instead of on. If `ni` works properly then `i6` should certainly not produce syntax errors, though it will surely produce warnings; all the same it can fail if, say, Z-machine memory limits are exceeded. The interface should deal with such failures exactly as it would in a non-Release run.

(5) `i6` having returned 0 to indicate success, the interface next calls `cblorb` as follows. Let `Path` be the path to the folder containing the Inform project being released, which we'll call `This.inform`. Then the interface should call:

```
cblorb -platform "Path/This.inform/Release.blurb" "Path/This.inform/Build/output.gblorb"
```

where `-platform` should be one of `-osx`, `-windows` or `-unix`. (The default is `-osx`.) The two filename arguments are the Blurb script for `cblorb` to follow, which was written by `ni` at step 3, and the filename of the Blorb file which it should write. Note that the interface should give this the extension ".gblorb" if the Glulx setting is in force, and ".zblorb" if the Z-machine.

(6) Like its predecessors, `cblorb` can produce error messages, so the interface must again look at the return code. The interface should display the Errors panel and, on the Problems tab, render:
   (a) `GoodCblorb.html`, with picture of wrapped parcel, if `cblorb` returned 0, or
   (b) `ErrorCblorb.html`, with picture of broken packaging, if not.

(7) There are no more tools to call, but the interface has one last duty (if `cblorb` succeeded) – to move the blorb somewhere sensible on disc, where the user can see it. Leaving it where it is will not do – the user never looks inside the Build project of a folder, which on Mac OS X, for instance, is not even visible. To see what to do, the interface must look at the textual output from `cblorb`, printed to `stdout` (of course the interface is free to redirect this if it wants to). If `cblorb` printed a line in the form:

```
Copy blorb to: [[...]]
```

then the interface should do as it's told. For instance:

```
Copy blorb to: [[/Users/gnelson/Examples/Bronze Materials/Release/Bronze.gblorb]]
```

If `cblorb` printed no such line, the interface should put up a Save As... dialogue box, and invite the user to choose a destination.

§**5.** `cblorb` **at the command line.** When using `cblorb` as a command-line tool, it's probably convenient to download a standalone copy from the Inform website, though that's identical to the copy squirreled away somewhere in the application. On Mac OS X, it lives at:

```
Inform.app/Contents/Resources/Compilers/cBlorb
```

Its main usage is:

```
cblorb -platform [-options] blurbfile [blorbfile]
```

where `-platform` should be one of `-osx`, `-windows`, `-unix`. At present the only practical difference this makes is that the Windows setting causes `cblorb` to use `\` instead of `/` as a filename separator.

The blorbfile filename is optional since `cblorb` does not always need to make a blorb; that depends on the instructions handed to it in the `blurbfile`.

§**6.** The other command-line options are:

`-help`: prints summaries of command-line use and the Blurb language.

`-trace`: mainly for debugging, but possibly also useful as a verbose mode.

`-project Whatever.inform`: tells `cblorb` to assume the usual settings for this project. (That means the blurbfile is set to `Whatever.inform/Release.blurb` and the blorbfile to `Whatever.inform/Build/output.gblorb`.)

§**7. Example blurb scripts.** This first script instructs `cblorb` to carry out its mission – it makes a simple Blorb wrapping up a story file with bibliographic data, but nothing more, and nothing else is released.

```
storyfile "/Users/gnelson/Examples/Zinc.inform/Build/output.ulx" include
ifiction "/Users/gnelson/Examples/Zinc.inform/Metadata.iFiction" include
```

These two lines tell `cblorb` to include the story file and the iFiction record respectively.

§**8.** A more ambitious Blorb can be made like so:

```
storyfile leafname "Audiophilia.gblorb"
storyfile "/Users/gnelson/Examples/Audiophilia.inform/Build/output.ulx" include
ifiction "/Users/gnelson/Examples/Audiophilia.inform/Metadata.iFiction" include
cover "/Users/gnelson/Examples/Audiophilia Materials/Cover.png"
picture 1 "/Users/gnelson/Examples/Audiophilia Materials/Cover.png"
sound 3 "/Users/gnelson/Examples/Audiophilia Materials/Sounds/Powermac.aiff"
sound 4 "/Users/gnelson/Examples/Audiophilia Materials/Sounds/Bach.ogg"
```

The cover image is included only once, but declaring it as picture 1 makes it available to the story file for display internally as well as externally. Resource ID 2, apparently skipped, is in fact the story file.

§**9.** And here's a very short script, which makes `cblorb` generate a solution file from the Skein of a project:

```
project folder "/Users/gnelson/Examples/Zinc.inform"
release to "/Users/gnelson/Examples/Zinc Materials/Release"
solution
```

This time no blorb file is made. The opening line tells `cblorb` which Inform project we're dealing with, allowing it to look at the various files inside – its Skein, for instance, which is used to create a solution. The second line tells `cblorb` where to put all of its output – everything it makes. Only the third line directly causes `cblorb` to do anything.

§**10.** More ambitiously, this time we'll make a website for a project, but again without making a blorb:

```
project folder "/Users/gnelson/Examples/Audiophilia.inform"
release to "/Users/gnelson/Examples/Audiophilia Materials/Release"
placeholder [IFID] = "AD5648BA-18A2-48A6-9554-4F6C53484824"
placeholder [RELEASE] = "1"
placeholder [YEAR] = "2009"
placeholder [TITLE] = "Audiophilia"
placeholder [AUTHOR] = "Graham Nelson"
placeholder [BLURB] = "A test project for sound effect production."
template path "/Users/gnelson/Library/Inform/Templates"
css
website "Standard"
```

The first novelty here is the setting of placeholders. These are named pieces of text which appear on the website being generated: where the text "[RELEASE]" appears in the template, `cblorb` writes the value we've set for it, in this case "1". Some of these values look like numbers, but to `cblorb` they all hold text. A few placeholder names are reserved by `cblorb` for its own use, and it will produce errors if we try to set those, but none of those in this example is reserved.

Template paths tell `cblorb` where to find templates. Any number of these can be set – including none at all, but if so then commands needing a named template, like `website`, can't be used. `cblorb` looks for any template it needs by trying each template path in turn (the earliest defined having the highest priority). The blurb files produced by `ni` in its `-release` mode containa chain of three template paths, for the individual project folder, the user's library of installed templates, and the built-in stock inside the Inform user interface application, respectively.

The command `css` tells `cblorb` that it is allowed to use CSS styles to make its web pages more appealing to look at: this results in generally better HTML, easier to use in other contexts, too.

All of that set things up so that the `website` command could be used, which actually does something – it creates a website in the release-to location, taking its design from the template named. If we were to add any of these commands –

```
source public
solution public
ifiction public
```

– then the website would be graced with these additions.

§**11.** The previous examples all involved Inform projects, but `cblorb` can also deal with stand-alone files of Inform source text – notably extensions. For example, here we make a website out of an extension:

```
release to "Test Site"
placeholder [TITLE] = "Locksmith"
placeholder [AUTHOR] = "Emily Short"
placeholder [RUBRIC] = "Implicit handling of doors and..." and so on
template path "/Users/gnelson/Library/Inform/Templates"
css
release file "style.css" from "Extended"
release file "index.html" from "Extended"
release file "Extensions/Emily Short/Locksmith.i7x"
release source "Extensions/Emily Short/Locksmith.i7x" using "extsrc.html" from "Extended"
```

This time we're using a template called "Extended", and the script tells `cblorb` exactly what to do with it. The "release file... from..." command tells `cblorb` to extract the named file from this template and to copy it into the release folder – if it's a ".html" file, placeholders are substituted with their values. The simpler form, "release file ...", just tells `cblorb` to copy that actual file – here, it puts a copy of the extension itself into the release folder. The final line produces a run of pages, in all likelihood, for the source and documentation of the extension, with the design drawn from "Extended" again.

("Extended" isn't supplied inside Inform; it's a template we're using to help generate the Inform website, rather than something meant for end users. There's nothing very special about it, in any case.)

§**12. Specification of the Blurb language.** A blorb script should be a text file, using the Unicode character set and encoded as UTF-8 without a byte order marker – in other words, a plain text file. It consists of lines of up to 10239 bytes in length each, divided by any of the four line-end markers in common use (`CR`, `LF`, `CR LF` or `LF CR`), though the same line-end marker should be used throughout the file.

Each command occupies one and only one line of text. (In Blorb 2001, the now-deprecated `palette` command could occupy multiple lines, but `cblorb` will choke on such a usage.) Lines are permitted to be empty or to contain only white space. Lines whose first non-white-space character is an exclamation mark are treated as comments, that is, ignored. "White space" means spaces and tab characters. An entirely empty blurb file, containing nothing but white space, is perfectly legal though useless.

In the following description:

⟨*string*⟩ means any text within double-quotes, not containing either double-quote or new-line characters, of up to 2048 bytes.

⟨*filename*⟩ means any double-quoted filename.

⟨*number*⟩ means a decimal number in the range 0 to 32767.

⟨*id*⟩ means either nothing at all, or a ⟨*number*⟩, or a sequence of up to 20 letters, digits or underscore characters `_`.

⟨*dim*⟩ indicates screen dimensions, and must take the form ⟨*number*⟩`x`⟨*number*⟩.

⟨*ratio*⟩ is a fraction in the form ⟨*number*⟩/⟨*number*⟩. 0/0 is legal but otherwise both numbers must be positive.

⟨*colour*⟩ is a colour expressed as six hexadecimal digits, as in some HTML tags: for instance `F5DEB3` is the colour of wheat, with red value `F5` (on a scale `00`, none, to `FF`, full), green value `DE` and blue value `B3`. Hexadecimal digits may be given in either upper or lower case.

§**13.** The full set of commands is as follows. First, core commands for making a blorb:

`author` ⟨*string*⟩

Adds this author name to the file.

`copyright` ⟨*string*⟩

Adds this copyright declaration to the blorb file. It would normally consist of short text such as "(c) J. Mango Pineapple 2007" rather than a lengthy legal discourse.

`release` ⟨*number*⟩

Gives this release number to the blorb file.

`auxiliary` ⟨*filename*⟩ ⟨*string*⟩

Tells us that an auxiliary file – for instance, a PDF manual – is associated with the release but will not be embedded directly into the blorb file. For instance,

```
auxiliary "map.png" "Black Pete's treasure map"
```

The string should be a textual description of the contents. Every auxiliary file should have a filename including an extension usefully describing its format, as in ".png": if there is no extension, then the auxiliary resource is assumed to be a mini-website housed in a subfolder with this name.

`ifiction` ⟨*filename*⟩ `include`

The file should be a valid iFiction record for the work. This is an XML file specified in the Treaty of Babel, a cross-IF-system standard for specifying bibliographic data; it will be embedded into the blorb.

`storyfile` ⟨*filename*⟩ *unsupported by* `cblorb`
`storyfile` ⟨*filename*⟩ `include`

Specifies the filename of the story file which these resources are being attached to. Blorb 2001 allowed for blorbs to be made which held everything to do with the release *except* the story file; that way a release might consist of one story file plus one Blorb file containing its pictures and sounds. The Blorb file would then contain a note of the release number, serial code and checksum of the associated story file so that an interpreter can try to match up the two files at run-time. If the `include` option is used, however, the entire

story file is embedded within the Blorb file, so that game and resources are all bound up in one single file. `cblorb` always does this, and does not support `storyfile` without `include`.

§**14.** Second, now-deprecated commands describing our ideal screen display:

`palette 16 bit` *unsupported by* `cblorb`
`palette 32 bit` *unsupported by* `cblorb`
`palette {` ⟨*colour-1*⟩ ⟨*colour-N*⟩ `}` *unsupported by* `cblorb`

Blorb allows designers to signal to the interpreter that a particular colour-scheme is in use. The first two options simply suggest that the pictures are best displayed using at least 16-bit, or 32-bit, colours. The third option specifies colours used in the pictures in terms of red/green/blue levels, and the braces allow the sequence of colours to continue over many lines. At least one and at most 256 colours may be defined in this way. This is only a "clue" to the interpreter; see the Blorb specification for details.

`resolution` ⟨*dim*⟩ *unsupported by* `cblorb`
`resolution` ⟨*dim*⟩ `min` ⟨*dim*⟩ *unsupported by* `cblorb`
`resolution` ⟨*dim*⟩ `max` ⟨*dim*⟩ *unsupported by* `cblorb`
`resolution` ⟨*dim*⟩ `min` ⟨*dim*⟩ `max` ⟨*dim*⟩ *unsupported by* `cblorb`

Allows the designer to signal a preferred screen size, in real pixels, in case the interpreter should have any choice over this. The minimum and maximum values are the extreme values at which the designer thinks the game will be playable: they're optional, the default values being 0×0 and $\infty \times \infty$.

§**15.** Third, commands for adding audiovisual resources:

`sound` ⟨*id*⟩ ⟨*filename*⟩
`sound` ⟨*id*⟩ ⟨*filename*⟩ `repeat` ⟨*number*⟩ *unsupported by* `cblorb`
`sound` ⟨*id*⟩ ⟨*filename*⟩ `repeat forever` *unsupported by* `cblorb`
`sound` ⟨*id*⟩ ⟨*filename*⟩ `music` *unsupported by* `cblorb`
`sound` ⟨*id*⟩ ⟨*filename*⟩ `song` *unsupported by* `cblorb`

Tells us to take a sound sample from the named file and make it the sound effect with the given number. Most forms of `sound` are now deprecated: repeat information (the number of repeats to be played) is meaningful only with Z-machine version 3 story files using sound effects, and Inform 7 does not generate those; the `music` and `song` keywords specify unusual sound formats. Nowadays the straight `sound` command should always be used regardless of format.

`picture` ⟨*id*⟩ ⟨*filename*⟩
`picture` ⟨*id*⟩ ⟨*filename*⟩ `scale` ⟨*ratio*⟩ *unsupported by* `cblorb`
`picture` ⟨*id*⟩ ⟨*filename*⟩ `scale min` ⟨*ratio*⟩ *unsupported by* `cblorb`
`picture` ⟨*id*⟩ ⟨*filename*⟩ `scale` ⟨*ratio*⟩ `min` ⟨*ratio*⟩ *unsupported by* `cblorb`

(and so on) is a similar command for images. In 2001, the image file was required to be a PNG, but it can now alternatively be a JPEG.

Optionally, the designer can specify a scale factor at which the interpreter will display the image – or, alternatively, a range of acceptable scale factors, from which the interpreter may choose its own scale factor. (By default an image is not scaleable and an interpreter must display it pixel-for-pixel.) There are three optional scale factors given: the preferred scale factor, the minimum and the maximum allowed. The minimum and maximum each default to the preferred value if not given, and the default preferred scale factor is 1. Scale factors are expressed as fractions: so for instance,

```
picture "flag/png" scale 3/1
```

means "always display three times its normal size", whereas

```
picture "backdrop/png" scale min 1/10 max 8/1
```

means"you can display this anywhere between one tenth normal size and eight times normal size, but if possible it ought to be just its normal size".

`cblorb` does not support any of the scaled forms of `picture`. As with the exotic forms of `sound`, they now seem passé. We no longer need to worry too much about the size of the blorb file, nor about screens with very low resolution; an iPhone today has a screen resolution close to that of a typical desktop of 2001.

`cover` ⟨*filename*⟩

specifies that this is the cover art; it must also be declared with a `picture` command in the usual way, and must have picture ID 1.

§**16.** Three commands help us to specify locations.

`project folder` ⟨*filename*⟩

Tells `cblorb` to look for associated resources, such as the Skein file, within this Inform project.

`release to` ⟨*filename*⟩

Tells `cblorb` that all of its output should go into this folder. (Well, except that the blorb file itself will be written to the location specified in the command line arguments, but see the description above of how `cblorb` then contrives to move it.) The folder must already exist, and `cblorb` won't create it. Under some circumstances Inform will seem to be creating the release folder if it doesn't already exist, but that's always the work of `ni`, not `cblorb`.

`template path` ⟨*filename*⟩

Sets a search path for templates – a folder in which to look for them. There can be any number of template paths set, and `cblorb` checks them in order of declaration (i.e., most important first).

§**17.** Next we come to commands for specifying what `cblorb` should release. At present it has six forms of output: Blorb file, solution file, source text, iFiction record, miscellaneous file and website.

No explicit single command causes a Blorb file to be generated; it will be made automatically if one of the above commands to include the story file, pictures, etc., is present in the script, and otherwise not generated.

```
solution
solution public
```

causes a solution file to be generated in the release folder. The mechanism for this is described in *Writing with Inform*. The difference between the two commands affects only a website also being made, if one is: a public solution will be included in its links, thus being made available to the public who read the website.

```
ifiction
ifiction public
```

is similar, but for the iFiction record of the project.

```
source
source public
```

is again similar, but here there's a twist. If the source is public, then `cblorb` doesn't just include it on a website: it generates multiple HTML pages to show it off in HTML form, as well as including the plain text original.

Miscellaneous files can be released like so:

`release file` ⟨*filename*⟩

Here `cblorb` acts as no more than a file-copy utility; a verbatim copy of the named file is placed in the release folder.

§**18.** Finally we come to web pages.

`css`

enables the use of CSS-defined styles within the HTML generated by `cblorb`. This has an especially marked effect when `cblorb` is generating HTML versions of Inform source text, and is *a good thing*. Unless there is reason not to, every blurb script generating websites ought to contain this command.

`release file` ⟨*filename*⟩ `from` ⟨*template*⟩

causes the named file to be found from the given template. If it can't be found in that template, `cblorb` tries to find it from a template called "Standard". If it isn't there either, or `cblorb` can't find any template called "Standard" in any of its template paths (see above), then an error message is produced. But if all goes well the file is copied into the release folder. If it has the file extension ".html" (in lower case, and using that exact form, i.e., not ".HTM" or some other variation) then any placeholders in the file will be expanded with their values. A few reserved placeholders have special effects, causing `cblorb` to expand interesting text in their places – see *Writing with Inform* for more on this.

`release source` ⟨*filename*⟩ using ⟨*filename*⟩ `from` ⟨*template*⟩

makes `cblorb` convert the Inform source text in the first filename into a suite of web pages using the style of the given file from the given template.

`website` ⟨*template*⟩

saves the best until last: it makes a complete website for an Inform project, using the named template. This means that the CSS file is copied into place (assuming `css` is used), the "index.html" is released from the template, the source of the project is run through `release source` using "source.html" from the template (assuming `source public` is used), and any extra files specified in the template's "(extras.txt)" are released as well. See *Writing with Inform* for more.

1 Services

`1/main`: *Main.w* To parse command-line arguments and take the necessary steps to obey them.

`1/mem`: *Memory.w* To allocate memory suitable for the dynamic creation of objects of different sizes, placing some larger objects automatically into doubly linked lists and assigning each a unique allocation ID number.

`1/text`: *Text Files.w* To read text files of whatever flavour, one line at a time.

`1/blurb`: *Blurb Parser.w* To read and follow the instructions in the blurb file, our main input.

Main 1/main

Purpose

To parse command-line arguments and take the necessary steps to obey them.

Definitions

¶1. We will need the following:

```
#include "stdlib.h"
#include "stdio.h"
#include "string.h"
#include "time.h"
#include "ctype.h"
```

¶2. We identify which platform we're running on thus:

```
define OSX_PLATFORM 1
define WINDOWS_PLATFORM 2
define UNIX_PLATFORM 3
```

¶3. Since we use flexible-sized memory allocation, `cblorb` contains few hard maxima on the size or complexity of its input, but:

```
define MAX_FILENAME_LENGTH 2048                 total length of pathname including leaf and extension
define MAX_EXTENSION_LENGTH 32                  extension part of filename, for auxiliary files
define MAX_VAR_NAME_LENGTH 32                   length of name of placeholder variable like "[AUTHOR]"
define MAX_TEXT_FILE_LINE_LENGTH 10240          for any single line in the project's source text
define MAX_SOURCE_TEXT_LINES 2000000000;        enough for 300 copies of the Linux kernel source – plenty!
```

¶4. Miscellaneous settings:

```
define VERSION "cBlorb 1.1"
define TRUE 1
define FALSE 0
```

¶5. Some global variables:

```
char SEP_CHAR = '/';                            set to the correct value for the platform by main()
int trace_mode = FALSE;                         print diagnostics to stdout while running?
int error_count = 0;                            number of error messages produced so far
int current_year_AD = 0;                        e.g., 2008
int blorb_file_size = 0;                        size in bytes of the blorb file written
int no_pictures_included = 0;                   number of picture resources included in the blorb
int no_sounds_included = 0;                     number of sound resources included in the blorb
int use_css_code_styles = FALSE;                use <span class="X"> markings when setting code
char project_folder[MAX_FILENAME_LENGTH];       pathname of I7 project folder, if any
char release_folder[MAX_FILENAME_LENGTH];       pathname of folder for website to write, if any
int cover_exists = FALSE;                       an image is specified as cover art
int cover_is_in_JPEG_format = TRUE;             as opposed to PNG format
```

§**1. Main.** Like most programs, this one parses command-line arguments, sets things up, reads the input and then writes the output.

That's a little over-simplified, though, because it also produces auxiliary outputs along the way, in the course of parsing the blurb file. The blorb file is only the main output – there might also be a web page and a solution file, for instance.

```
int main(int argc, char *argv[]) {
    int platform, produce_help;
    char blurb_filename[MAX_FILENAME_LENGTH];
    char blorb_filename[MAX_FILENAME_LENGTH];
    ⟨Make the default settings 2⟩;
    ⟨Parse command-line arguments 3⟩;
    start_memory();
    establish_time();
    initialise_placeholders();
    print_banner();
    if (produce_help) { ⟨Produce help 4⟩; return 0; }
    parse_blurb_file(blurb_filename);
    write_blorb_file(blorb_filename);
    create_requested_material();
    print_report();
    free_memory();
    return 0;
}
```

The function main is where execution begins.

§**2.**

⟨Make the default settings 2⟩ ≡

```
    platform = OSX_PLATFORM;
    produce_help = FALSE;
    release_folder[0] = 0;
    project_folder[0] = 0;
    strcpy(blurb_filename, "Release.blurb");
    strcpy(blorb_filename, "story.zblorb");
```

This code is used in §1.

§3.

⟨Parse command-line arguments 3⟩ ≡

```
int arg, names = FALSE;
for (arg = 1, names = 0; arg < argc; arg++) {
    char *p = argv[arg];
    if (strlen(p) >= MAX_FILENAME_LENGTH) {
        fprintf(stderr, "cblorb: command line argument %d too long\n", arg+1);
        return 1;
    }
    if (strcmp(p, "-help") == 0) { produce_help = TRUE; continue; }
    if (strcmp(p, "-osx") == 0) { platform = OSX_PLATFORM; continue; }
    if (strcmp(p, "-windows") == 0) { platform = WINDOWS_PLATFORM; continue; }
    if (strcmp(p, "-unix") == 0) { platform = UNIX_PLATFORM; continue; }
    if (strcmp(p, "-trace") == 0) { trace_mode = TRUE; continue; }
    if (strcmp(p, "-project") == 0) {
        arg++; if (arg == argc) ⟨Command line syntax error 5⟩;
        strcpy(project_folder, argv[arg]);
        continue;
    }
    if (p[0] == '-') ⟨Command line syntax error 5⟩;
    names++;
    switch (names) {
        case 1: strcpy(blurb_filename, p); break;
        case 2: strcpy(blorb_filename, p); break;
        default: ⟨Command line syntax error 5⟩;
    }
}
if (platform == WINDOWS_PLATFORM) SEP_CHAR = '\\'; else SEP_CHAR = '/';
if (project_folder[0] != 0) {
    if (names > 0) ⟨Command line syntax error 5⟩;
    sprintf(blurb_filename, "%s%cRelease.blurb", project_folder, SEP_CHAR);
    sprintf(blorb_filename, "%s%cBuild%coutput.zblorb", project_folder, SEP_CHAR, SEP_CHAR);
}
if (trace_mode)
    printf("! Blurb in: <%s>\n! Blorb out: <%s>\n",
        blurb_filename, blorb_filename);
```

This code is used in §1.

§4.

⟨Produce help 4⟩ ≡

```
printf("This is cblorb, a component of Inform 7 for packaging up IF materials.\n\n");
⟨Show command line usage 6⟩;
summarise_blurb();
```

This code is used in §1.

§**5.**

⟨Command line syntax error 5⟩ ≡
```
    ⟨Show command line usage 6⟩;
    return 1;
```
This code is used in §3.

§**6.**

⟨Show command line usage 6⟩ ≡
```
    printf("usage: cblorb -platform [-options] [blurbfile [blorbfile]]\n\n");
    printf("  Where -platform should be -osx (default), -windows, or -unix\n");
    printf("  As an alternative to giving filenames for the blurb and blorb,\n");
    printf("    -project Whatever.inform\n");
    printf("  sets blurbfile and blorbfile names to the natural choices.\n");
    printf("  The other possible options are:\n");
    printf("    -help ... print this usage summary\n");
    printf("    -trace ... print diagnostic information during run\n");
```
This code is used in §4,5.

§**7. Time.** It wouldn't be a tremendous disaster if the host OS had no access to an accurate time of day, in fact.

```
time_t the_present;
struct tm *here_and_now;
void establish_time(void) {
    the_present = time(NULL);
    here_and_now = localtime(&the_present);
}
```

§**8.** The placeholder variable [YEAR] is initialised to the year in which `cBlorb` runs, according to the host operating system, at least. (It can of course then be overridden by commands in the blurb file, and Inform always does this in the blurb files it writes. But it leaves [DATESTAMP] and [TIMESTAMP] alone.)

```
void initialise_time_variables(void) {
    char datestamp[100], infocom[100], timestamp[100];
    char *weekdays[] = { "Sunday", "Monday", "Tuesday", "Wednesday",
        "Thursday", "Friday", "Saturday" };
    char *months[] = { "January", "February", "March", "April", "May", "June",
        "July", "August", "September", "October", "November", "December" };
    set_placeholder_to_number("YEAR", here_and_now->tm_year+1900);
    sprintf(datestamp, "%s %d %s %d", weekdays[here_and_now->tm_wday],
        here_and_now->tm_mday, months[here_and_now->tm_mon], here_and_now->tm_year+1900);
    sprintf(infocom, "%02d%02d%02d",
        here_and_now->tm_year-100, here_and_now->tm_mon + 1, here_and_now->tm_mday);
    sprintf(timestamp, "%02d:%02d.%02d", here_and_now->tm_hour,
        here_and_now->tm_min, here_and_now->tm_sec);
    set_placeholder_to("DATESTAMP", datestamp, 0);
    set_placeholder_to("INFOCOMDATESTAMP", infocom, 0);
    set_placeholder_to("TIMESTAMP", timestamp, 0);
}
```
The function initialise_time_variables is called from 3/place.

§**9. Opening and closing banners.** Note that `cBlorb` customarily prints informational messages with an initial `!`, so that the piped output from `cBlorb` could be used as an `Include` file in I6 code; that isn't in fact how I7 uses `cBlorb`, but it's traditional for blorbing programs to do this.

```
void print_banner(void) {
    printf("! %s [executing on %s at %s]\n",
        VERSION, read_placeholder("DATESTAMP"), read_placeholder("TIMESTAMP"));
    printf("! The blorb spell (safely protect a small object ");
    printf("as though in a strong box).\n");
}
```

§**10.** And then at the end:

```
void print_report(void) {
    if (error_count > 0) {
        printf("! Completed: %d error(s)\n", error_count);
        exit(1);
    }
    if (blorb_file_size > 0) {
        printf("! Completed: wrote blorb file of size %d bytes ", blorb_file_size);
        printf("(%d picture(s), %d sound(s))\n", no_pictures_included, no_sounds_included);
    } else {
        printf("! Completed: no blorb output requested\n");
    }
}
```

Memory 1/mem

Purpose

To allocate memory suitable for the dynamic creation of objects of different sizes, placing some larger objects automatically into doubly linked lists and assigning each a unique allocation ID number.

Definitions

¶1. This section is slightly simplified, but essentially copied, from the memory allocator used in the main Inform 7 compiler.

It allocates memory as needed to store the numerous objects of different sizes, all typedef'd structs. There's no garbage collection because nothing is ever destroyed. Each type has its own doubly-linked list, and in each type the objects created are given unique IDs (within that type) counting upwards from 0.

¶2. Before going much further, we will need to anticipate what the memory manager wants. In order to keep the doubly linked lists and the allocation ID, every structure subject to this regime will need extra elements holding the necessary links and ID number. We define these elements with a macro (concealing its meaning from all other sections).

Smaller objects are stored in arrays, and their structure declarations do not use the following macro.

```
define MEMORY_MANAGEMENT
    int allocation_id;                    Numbered from 0 upwards in creation order
    void *next_structure;                 Next object in double-linked list
    void *prev_structure;                 Previous object in double-linked list
```

¶3. There is no significance to the order in which structures are registered with the memory system, but `NO_MEMORY_TYPES` must be 1 more than the highest MT number, so do not add to this list without incrementing it. There can in principle be up to 1000 memory types.

```
define auxiliary_file_MT 0
define skein_node_MT 1
define chunk_metadata_MT 2
define placeholder_MT 3
define heading_MT 4
define table_MT 5
define segment_MT 6
define request_MT 7
define template_MT 8
define template_path_MT 9
define NO_MEMORY_TYPES 10             must be 1 more than the highest _MT constant above
```

---

§**1.** For each type of object to be allocated, a single structure of the following design is maintained. Types which are allocated individually, like world objects, have `no_allocated_together` set to 1, and the doubly linked list is of the objects themselves. For types allocated in small arrays (typically of 100 objects at a time), `no_allocated_together` is set to the number of objects in each completed array (so, typically 100) and the doubly linked list is of the arrays.

```
typedef struct allocation_status_structure {
    actually needed for allocation purposes:
    int objects_allocated;                    total number of objects (or arrays) ever allocated
    void *first_in_memory;                                        head of doubly linked list
    void *last_in_memory;                                         tail of doubly linked list
    used only to provide statistics for the debugging log:
    char *name_of_type;                                           e.g., "lexicon_entry_MT"
    int bytes_allocated;                  total allocation for this type of object, not counting overhead
    int objects_count;                            total number currently in existence (i.e., undeleted)
    int no_allocated_together;                  number of objects in each array of this type of object
} allocation_status_structure;
```

The structure allocation_status_structure is private to this section.

§**2.** The memory allocator itself needs some memory, but only a fixed-size and fairly small array of the structures defined above. The allocator can safely begin as soon as this is initialised.

```
allocation_status_structure alloc_status[NO_MEMORY_TYPES];

void start_memory(void) {
    int i;
    for (i=0; i<NO_MEMORY_TYPES; i++) {
        alloc_status[i].first_in_memory = NULL;
        alloc_status[i].last_in_memory = NULL;
        alloc_status[i].objects_allocated = 0;
        alloc_status[i].objects_count = 0;
        alloc_status[i].bytes_allocated = 0;
        alloc_status[i].no_allocated_together = 1;
        alloc_status[i].name_of_type = "unused";
    }
}
```

The function start_memory is called from 1/main.

§**3. Architecture.** The memory manager is built in three levels, with its interface to the rest of `cblorb` being entirely at level 3 (except that when it shuts down it calls a level 1 routine to free everything). Each level uses the one below it.

(3) Managing linked lists of large objects, within which objects can be created at any point, and from which objects can be deleted; and providing a way to create new small objects of any given type.
(2) Allocating some thousands of memory frames, each holding one large object or an array of small objects.
(1) Allocating and freeing a few dozen large blocks of contiguous memory.

§**4. Level 1: memory blocks.** Memory is allocated in blocks of 100K, within which objects are allocated as needed. The "safety margin" is the number of spare bytes left blank at the end of each object: this is done because we want to be paranoid about compilers on different architectures aligning structures to different boundaries (multiples of 4, 8, 16, etc.). Each block also ends with a firebreak of zeroes, which ought never to be touched: we want to minimise the chance of a mistake causing a memory exception which crashes the compiler, because if that happens it will be difficult to recover the circumstances from the debugging log.

```
define SAFETY_MARGIN 64
define BLANK_END_SIZE 128
```

§**5.** At present `MEMORY_GRANULARITY` is 100K. This is the quantity of memory allocated by each individual `malloc` call.

After `MAX_BLOCKS_ALLOWED` blocks, we throw in the towel: we must have fallen into an endless loop which creates endless new objects somewhere. (If this ever happens, it would be a bug: the point of this mechanism is to be able to recover. Without this safety measure, OS X in particular would grind slowly to a halt, never refusing a `malloc`, until the user was unable to get the GUI responsive enough to kill the process.)

```
define MAX_BLOCKS_ALLOWED 15000
define MEMORY_GRANULARITY 100*1024*4                    which must be divisible by 1024

int no_blocks_allocated = 0;
int total_objects_allocated = 0;                        a much larger number, used only for the debugging log
```

§**6.** Memory blocks are stored in a linked list, and we keep track of the size of the current block: that is, the block at the tail of the list. Each memory block consists of a header structure, followed by `SAFETY_MARGIN` null bytes, followed by actual data.

```
typedef struct memblock_header {
    int block_number;
    struct memblock_header *next;
    char *the_memory;
} memblock_header;

memblock_header *first_memblock_header = NULL;          head of list of memory blocks
memblock_header *current_memblock_header = NULL;        tail of list of memory blocks
int used_in_current_memblock = 0;                       number of bytes so far used in the tail memory block
```

The structure memblock_header is private to this section.

§**7.** The actual allocation and deallocation is performed by the following pair of routines.

```
void allocate_another_block(void) {
    unsigned char *cp;
    memblock_header *mh;
    ⟨Allocate and zero out a block of memory, making cp point to it 8⟩;
    mh = (memblock_header *) cp;
    used_in_current_memblock = sizeof(memblock_header) + SAFETY_MARGIN;
    mh->the_memory = (void *) (cp + used_in_current_memblock);
    ⟨Add new block to the tail of the list of memory blocks 9⟩;
}
```

§**8.** Note that `cp` and `mh` are set to the same value: they merely have different pointer types as far as the C compiler is concerned.

⟨Allocate and zero out a block of memory, making cp point to it 8⟩ ≡

```
int i;
if (no_blocks_allocated++ >= MAX_BLOCKS_ALLOWED)
    fatal(
        "the memory manager has halted cblorb, which seems to be generating "
        "endless structures. Presumably it is trapped in a loop");
check_memory_integrity();
cp = (unsigned char *) (malloc(MEMORY_GRANULARITY));
if (cp == NULL) fatal("Run out of memory: malloc failed");
for (i=0; i<MEMORY_GRANULARITY; i++) cp[i] = 0;
```

This code is used in §7.

§**9.** As can be seen, memory block numbers count upwards from 0 in order of their allocation.

⟨Add new block to the tail of the list of memory blocks 9⟩ ≡

```
if (current_memblock_header == NULL) {
    mh->block_number = 0;
    first_memblock_header = mh;
} else {
    mh->block_number = current_memblock_header->block_number + 1;
    current_memblock_header->next = mh;
}
current_memblock_header = mh;
```

This code is used in §7.

§**10.** Freeing all this memory again is just a matter of freeing each block in turn, but of course being careful to avoid following links in a just-freed block.

```
void free_memory(void) {
    memblock_header *mh = first_memblock_header;
    while (mh != NULL) {
        memblock_header *next_mh = mh->next;
        void *p = (void *) mh;
        free(p);
        mh = next_mh;
    }
}
```

The function free_memory is called from 1/main.

§**11. Level 2: memory frames and integrity checking.** Within these extensive blocks of contiguous memory, we place the actual objects in between "memory frames", which are only used at present to police the integrity of memory: again, finding obscure and irritating memory-corruption bugs is more important to us than saving bytes. Each memory frame wraps either a single large object, or a single array of small objects.

```
define INTEGRITY_NUMBER 0x12345678                    a value unlikely to be in memory just by chance

typedef struct memory_frame {
    int integrity_check;                              this should always contain the INTEGRITY_NUMBER
    struct memory_frame *next_frame;                  next frame in the list of memory frames
    int mem_type;                                     type of object stored in this frame
    int allocation_id;                                allocation ID number of object stored in this frame
} memory_frame;
```

The structure memory_frame is private to this section.

§**12.** There is a single linked list of all the memory frames, perhaps of about 10000 entries in length, beginning here. (These frames live in different memory blocks, but we don't need to worry about that.)

```
memory_frame *first_memory_frame = NULL;              earliest memory frame ever allocated
memory_frame *last_memory_frame = NULL;               most recent memory frame allocated
```

§**13.** If the integrity numbers of every frame are still intact, then it is pretty unlikely that any bug has caused memory to overwrite one frame into another. `check_memory_integrity` might on very large runs be run often, if we didn't prevent this: since the number of calls would be roughly proportional to memory usage, we would implicitly have an $O(n^2)$ running time in the amount of storage n allocated.

```
int calls_to_cmi = 0;
void check_memory_integrity(void) {
    int c;
    memory_frame *mf;
    c = calls_to_cmi++;
    if (!((c<10) || (c == 100) || (c == 1000) || (c == 10000))) return;
    for (c = 0, mf = first_memory_frame; mf; c++, mf = mf->next_frame)
        if (mf->integrity_check != INTEGRITY_NUMBER)
            fatal("Memory manager failed integrity check");
}
void debug_memory_frames(int from, int to) {
    int c;
    memory_frame *mf;
    for (c = 0, mf = first_memory_frame; (mf) && (c <= to); c++, mf = mf->next_frame)
        if (c >= from) {
            char *desc = "corrupt";
            if (mf->integrity_check == INTEGRITY_NUMBER)
                desc = alloc_status[mf->mem_type].name_of_type;
        }
}
```

§**14.** We have seen how memory is allocated in large blocks, and that a linked list of memory frames will live inside those blocks; we have seen how the list is checked for integrity; but we not seen how it is built. Every memory frame is created by the following function:

```
void *allocate_mem(int mem_type, int extent) {
    unsigned char *cp;
    memory_frame *mf;
    int bytes_free_in_current_memblock, extent_without_overheads = extent;
    extent += sizeof(memory_frame);                    each allocation is preceded by a memory frame
    extent += SAFETY_MARGIN;                  each allocation is followed by SAFETY_MARGIN null bytes
    ⟨Ensure that the current memory block has room for this many bytes 15⟩;
    cp = ((unsigned char *) (current_memblock_header->the_memory)) + used_in_current_memblock;
    used_in_current_memblock += extent;
    mf = (memory_frame *) cp;                                              the new memory frame,
    cp = cp + sizeof(memory_frame);                        following which is the actual allocated data
    mf->integrity_check = INTEGRITY_NUMBER;
    mf->allocation_id = alloc_status[mem_type].objects_allocated;
    mf->mem_type = mem_type;
    ⟨Add the new memory frame to the big linked list of all frames 16⟩;
    ⟨Update the allocation status for this type of object 17⟩;
    total_objects_allocated++;
    return (void *) cp;
}
```

§**15.** The granularity error below will be triggered the first time a particular object type is allocated. So this is not a potential time-bomb just waiting for a user with a particularly long and involved source text to discover.

⟨Ensure that the current memory block has room for this many bytes 15⟩ ≡

```
if (current_memblock_header == NULL) allocate_another_block();
bytes_free_in_current_memblock = MEMORY_GRANULARITY - (used_in_current_memblock + extent);
if (bytes_free_in_current_memblock < BLANK_END_SIZE) {
    allocate_another_block();
    if (extent+BLANK_END_SIZE >= MEMORY_GRANULARITY)
        fatal("Memory manager failed because granularity too low");
}
```

This code is used in §14.

§**16.** New memory frames are added to the tail of the list:

⟨Add the new memory frame to the big linked list of all frames 16⟩ ≡

```
mf->next_frame = NULL;
if (first_memory_frame == NULL) first_memory_frame = mf;
else last_memory_frame->next_frame = mf;
last_memory_frame = mf;
```

This code is used in §14.

§**17.** See the definition of `alloc_status` above.

⟨Update the allocation status for this type of object 17⟩ ≡

```
if (alloc_status[mem_type].first_in_memory == NULL)
    alloc_status[mem_type].first_in_memory = (void *) cp;
alloc_status[mem_type].last_in_memory = (void *) cp;
alloc_status[mem_type].objects_allocated++;
alloc_status[mem_type].bytes_allocated += extent_without_overheads;
```

This code is used in §14.

§**18. Level 3: managing linked lists of allocated objects.** We define macros which look as if they are functions, but for which one argument is the name of a type: expanding these macros provides suitable C functions to handle each possible type. These macros provide the interface through which all other sections of `cblorb` allocate and leaf through memory.

Note that `inweb` allows multi-line macro definitions without backslashes to continue them, unlike ordinary C. Otherwise these are "standard" macros, though this was my first brush with the `##` concatenation operator: basically `CREATE(thing)` expands into `(allocate_thing())` because of the `##`. (See Kernighan and Ritchie, section 4.11.2.)

```
define CREATE(type_name) (allocate_##type_name())
define CREATE_BEFORE(existing, type_name) (allocate_##type_name##_before(existing))
define DESTROY(this, type_name) (deallocate_##type_name(this))
define FIRST_OBJECT(type_name) ((type_name *) alloc_status[type_name##_MT].first_in_memory)
define LAST_OBJECT(type_name) ((type_name *) alloc_status[type_name##_MT].last_in_memory)
define NEXT_OBJECT(this, type_name) ((type_name *) (this->next_structure))
define PREV_OBJECT(this, type_name) ((type_name *) (this->prev_structure))
define NUMBER_CREATED(type_name) (alloc_status[type_name##_MT].objects_count)
```

§**19.** The following macros are widely used (well, the first one is, anyway) for looking through the double linked list of existing objects of a given type.

```
define LOOP_OVER(var, type_name)
    for (var=FIRST_OBJECT(type_name); var != NULL; var = NEXT_OBJECT(var, type_name))
define LOOP_BACKWARDS_OVER(var, type_name)
    for (var=LAST_OBJECT(type_name); var != NULL; var = PREV_OBJECT(var, type_name))
```

§**20. Allocator functions created by macros.** The following macros generate a family of systematically named functions. For instance, we shall shortly expand `ALLOCATE_INDIVIDUALLY(parse_node)`, which will expand to three functions: `allocate_parse_node`, `deallocate_parse_node` and `allocate_parse_node_before`.

Quaintly, `#type_name` expands into the value of `type_name` put within double-quotes.

```
define NEW_OBJECT(type_name) ((type_name *) allocate_mem(type_name##_MT, sizeof(type_name)))
define ALLOCATE_INDIVIDUALLY(type_name)
type_name *allocate_##type_name(void) {
    alloc_status[type_name##_MT].name_of_type = #type_name;
    type_name *prev_obj = LAST_OBJECT(type_name);
    type_name *new_obj = NEW_OBJECT(type_name);
    new_obj->allocation_id = alloc_status[type_name##_MT].objects_allocated-1;
    new_obj->next_structure = NULL;
    if (prev_obj != NULL)
        prev_obj->next_structure = (void *) new_obj;
    new_obj->prev_structure = prev_obj;
    alloc_status[type_name##_MT].objects_count++;
    return new_obj;
}
void deallocate_##type_name(type_name *kill_me) {
    type_name *prev_obj = PREV_OBJECT(kill_me, type_name);
    type_name *next_obj = NEXT_OBJECT(kill_me, type_name);
    if (prev_obj == NULL) {
        alloc_status[type_name##_MT].first_in_memory = next_obj;
    } else {
        prev_obj->next_structure = next_obj;
    }
    if (next_obj == NULL) {
        alloc_status[type_name##_MT].last_in_memory = prev_obj;
    } else {
        next_obj->prev_structure = prev_obj;
    }
    alloc_status[type_name##_MT].objects_count--;
}
type_name *allocate_##type_name##_before(type_name *existing) {
    type_name *new_obj = allocate_##type_name();
    deallocate_##type_name(new_obj);
    new_obj->prev_structure = existing->prev_structure;
    if (existing->prev_structure != NULL)
        ((type_name *) existing->prev_structure)->next_structure = new_obj;
    else alloc_status[type_name##_MT].first_in_memory = (void *) new_obj;
    new_obj->next_structure = existing;
    existing->prev_structure = new_obj;
    alloc_status[type_name##_MT].objects_count++;
    return new_obj;
}
```

§**21.** `ALLOCATE_IN_ARRAYS` is still more obfuscated. When we `ALLOCATE_IN_ARRAYS(X, 100)`, the result will be definitions of a new type `X_block` and functions `allocate_X`, `allocate_X_block`, `deallocate_X_block` and `allocate_X_block_before` (though the last is not destined ever to be used). Note that we are not provided with the means to deallocate individual objects this time: that's the trade-off for allocating in blocks.

```
define ALLOCATE_IN_ARRAYS(type_name, NO_TO_ALLOCATE_TOGETHER)
typedef struct type_name##_array {
    int used;
    struct type_name array[NO_TO_ALLOCATE_TOGETHER];
    MEMORY_MANAGEMENT
} type_name##_array;
ALLOCATE_INDIVIDUALLY(type_name##_array)
type_name##_array *next_##type_name##_array = NULL;
struct type_name *allocate_##type_name(void) {
    if ((next_##type_name##_array == NULL) ||
        (next_##type_name##_array->used >= NO_TO_ALLOCATE_TOGETHER)) {
        alloc_status[type_name##_array_MT].no_allocated_together = NO_TO_ALLOCATE_TOGETHER;
        next_##type_name##_array = allocate_##type_name##_array();
        next_##type_name##_array->used = 0;
    }
    return &(next_##type_name##_array->array[
        next_##type_name##_array->used++]);
}
```

The structure type_name##_array is private to this section.

§**22. Expanding many macros.** Each given structure must have a typedef name, say `marvel`, and can be used in one of two ways. Either way, we can obtain a new one with the macro `CREATE(marvel)`.

Either (a) it will be individually allocated. In this case `marvel_MT` should be defined with a new MT (memory type) number, and the macro `ALLOCATE_INDIVIDUALLY(marvel)` should be expanded. The first and last objects created will be `FIRST_OBJECT(marvel)` and `LAST_OBJECT(marvel)`, and we can proceed either way through a double linked list of them with `PREV_OBJECT(mv, marvel)` and `NEXT_OBJECT(mv, marvel)`. For convenience, we can loop through marvels, in creation order, using `LOOP_OVER(var, marvel)`, which expands to a `for` loop in which the variable `var` runs through each created marvel in turn; or equally we can run backwards through using `LOOP_BACKWARDS_OVER(var, marvel)`. In addition, there are corruption checks to protect the memory from overrunning accidents, and the structure can be used as a value in the symbols table. Good for large structures with significant semantic content.

Or (b) it will be allocated in arrays. Once again we can obtain new marvels with `CREATE(marvel)`. This is more efficient both in speed and memory usage, but we lose the ability to loop through the objects. For this arrangement, define `marvel_array_MT` with a new MT number and expand the macro `ALLOCATE_IN_ARRAYS(marvel, 100)`, where 100 (or what may you) is the number of objects allocated jointly as a block. Good for small structures used in the lower levels.

Here goes, then.

```
ALLOCATE_INDIVIDUALLY(auxiliary_file)
ALLOCATE_INDIVIDUALLY(skein_node)
ALLOCATE_INDIVIDUALLY(chunk_metadata)
ALLOCATE_INDIVIDUALLY(placeholder)
ALLOCATE_INDIVIDUALLY(heading)
ALLOCATE_INDIVIDUALLY(table)
ALLOCATE_INDIVIDUALLY(segment)
ALLOCATE_INDIVIDUALLY(request)
ALLOCATE_INDIVIDUALLY(template)
ALLOCATE_INDIVIDUALLY(template_path)
```

Text Files 1/text

Purpose

To read text files of whatever flavour, one line at a time.

1/text.§1-3 Text file positions; §4 Error messages; §5-10 File handling; §11-13 Two string utilities; §14 Other file utilities

---

Definitions

¶1.

```
typedef struct text_file_position {
    char text_file_filename[MAX_FILENAME_LENGTH];
    int line_count;
    int line_position;
    int skip_terminator;
    int actively_scanning;                    whether we are still interested in the rest of the file
} text_file_position;
```

The structure text_file_position is private to this section.

---

§1. **Text file positions.** This is useful for error messages:

```
void describe_file_position(text_file_position *tfp) {
    if (tfp == NULL) return;
    fprintf(stderr, "%s, line %d: ", tfp->text_file_filename, tfp->line_count);
}
```

The function describe_file_position is.

§2.

```
int tfp_get_line_count(text_file_position *tfp) {
    if (tfp == NULL) return 0;
    return tfp->line_count;
}
```

The function tfp_get_line_count is called from 1/blurb and 3/web.

§3.

```
void tfp_lose_interest(text_file_position *tfp) {
    tfp->actively_scanning = FALSE;
}
```

The function tfp_lose_interest is called from 3/web.

§4. **Error messages.** `cBlorb` is only minimally helpful when diagnosing problems, because it's intended to be used as the back end of a system which only generates correct blurb files, so that everything will work – ideally, the Inform user will never know that `cBlorb` exists.

```
text_file_position *error_position = NULL;
void set_error_position(text_file_position *tfp) {
    error_position = tfp;
}
void error(char *erm) {
    describe_file_position(error_position);
fprintf(stderr, "Error: %s\n", erm);
    error_count++;
}
void error_1(char *erm, char *s) {
    describe_file_position(error_position);
    fprintf(stderr, "Error: %s: '%s'\n", erm, s);
    error_count++;
}
void fatal(char *erm) {
    describe_file_position(error_position);
    fprintf(stderr, "Fatal error: %s\n", erm);
    exit(1);
}
void fatal_fs(char *erm, char *fn) {
    describe_file_position(error_position);
    fprintf(stderr, "Fatal error: %s: filename '%s'\n", erm, fn);
    exit(1);
}
```

The function set_error_position is called from 1/blurb.
The function error is called from 1/main, 1/blurb, 3/sol, 3/links and 3/place.
The function error_1 is called from 1/blurb, 3/rel, 3/templ and 3/web.
The function fatal is called from 1/mem, 1/blurb, 2/blorb and 3/web.
The function fatal_fs is called from 2/blorb and 3/sol.

§5. **File handling.** We read lines in, delimited by any of the standard line-ending characters, and send them one at a time to a function called `iterator`.

```
void file_read(char *filename, char *message, int serious,
    void (iterator)(char *, text_file_position *), text_file_position *start_at) {
    FILE *HANDLE;
    text_file_position tfp;
    ⟨Open the text file 6⟩;
    ⟨Set the initial position, seeking it in the file if need be 7⟩;
    ⟨Read in lines and send them one by one to the iterator 8⟩;
    fclose(HANDLE);
}
```

The function file_read is called from 1/blurb, 3/rel, 3/sol and 3/web.

§**6.**

⟨Open the text file 6⟩ ≡

```
if (strlen(filename) >= MAX_FILENAME_LENGTH) {
    if (serious) fatal_fs("filename too long", filename);
    error_1("filename too long", filename);
    return;
}
HANDLE = fopen(filename, "r");
if (HANDLE == NULL) {
    if (message == NULL) return;
    if (serious) fatal_fs(message, filename);
    else { error_1(message, filename); return; }
}
```

This code is used in §5.

§**7.** The ANSI definition of `ftell` and `fseek` says that, with text files, the only definite position value is 0 – meaning the beginning of the file – and this is what we initialise `line_position` to. We must otherwise only write values returned by `ftell` into this field.

⟨Set the initial position, seeking it in the file if need be 7⟩ ≡

```
if (start_at == NULL) {
    tfp.line_count = 1;
    tfp.line_position = 0;
    tfp.skip_terminator = 'X';
} else {
    tfp = *start_at;
    if (fseek(HANDLE, (long int) (tfp.line_position), SEEK_SET)) {
        if (serious) fatal_fs("unable to seek position in file", filename);
        error_1("unable to seek position in file", filename);
        return;
    }
}
tfp.actively_scanning = TRUE;
strcpy(tfp.text_file_filename, filename);
```

This code is used in §5.

§**8.** We aim to get this right whether the lines are terminated by `0A`, `0D`, `0A 0D` or `0D 0A`. The final line is not required to be terminated.

⟨Read in lines and send them one by one to the iterator 8⟩ ≡

```
char line[MAX_TEXT_FILE_LINE_LENGTH+1];
int i = 0, c = ' ';
while ((c != EOF) && (tfp.actively_scanning)) {
    c = fgetc(HANDLE);
    if ((c == EOF) || (c == '\x0a') || (c == '\x0d')) {
        line[i] = 0;
        if ((i > 0) || (c != tfp.skip_terminator)) {
            ⟨Feed the completed line to the iterator routine 9⟩;
            if (c == '\x0a') tfp.skip_terminator = '\x0d';
            if (c == '\x0d') tfp.skip_terminator = '\x0a';
        } else tfp.skip_terminator = 'X';
        ⟨Update the text file position 10⟩;
        i = 0;
    } else {
        if (i < MAX_TEXT_FILE_LINE_LENGTH) line[i++] = (char) c;
        else {
            if (serious) fatal_fs("line too long", filename);
            error_1("line too long (truncating it)", filename);
        }
    }
}
if ((i > 0) && (tfp.actively_scanning))
    ⟨Feed the completed line to the iterator routine 9⟩;
```

This code is used in §5.

§**9.** We update the line counter only when a line is actually sent:

⟨Feed the completed line to the iterator routine 9⟩ ≡

```
iterator(line, &tfp);
tfp.line_count++;
```

This code is used in §8.

§**10.** But we update the text file position after every apparent line terminator. This is because we might otherwise, on a Windows text file, end up with an `ftell` position in between the `CR` and the `LF`; if we resume at that point, later on, we'll then have an off-by-one error in the line numbering in the resumption as compared to during the original pass.

Properly speaking, `ftell` returns a long `int`, not an `int`, but on a 32-bit integer machine – which Inform requires – this gives us room for files to run to 2GB. Text files seldom come that large.

⟨Update the text file position 10⟩ ≡

```
tfp.line_position = (int) (ftell(HANDLE));
if (tfp.line_position == -1) {
    if (serious) fatal_fs("unable to determine position in file", filename);
    error_1("unable to determine position in file", filename);
}
```

This code is used in §8.

§11. Two string utilities.

```
char *trim_white_space(char *original) {
    int i;
    for (i=0; white_space(original[i]); i++) ;
    original += i;
    for (i=strlen(original)-1; ((i>=0) && (white_space(original[i]))); i--)
        original[i] = 0;
    return original;
}
```

The function trim_white_space is called from 1/blurb and 3/rel.

§12.

```
void extract_word(char *fword, char *line, int size, int word) {
    int i = 0;
    fword[0] = 0;
    while (word > 0) {
        word--;
        while (white_space(line[i])) i++;
        int j = 0;
        while ((line[i]) && (!white_space(line[i]))) {
            if (j < size-1) fword[j++] = tolower(line[i]);
            i++;
        }
        fword[j] = 0;
        if (line[i] == 0) break;
    }
    if (word > 0) fword[0] = 0;
}
```

The function extract_word is called from 3/web.

§13. Where we define white space as spaces and tabs only:

```
int white_space(int c) { if ((c == ' ') || (c == '\t')) return TRUE; return FALSE; }
```

§14. Other file utilities. Although this section is called "Text Files", it also has a couple of general-purpose file utilities:

```
char *get_filename_extension(char *filename) {
    int i = strlen(filename) - 1;
    while ((i>=0) && (filename[i] != '.') && (filename[i] != SEP_CHAR)) i--;
    if ((i<0) || (filename[i] == SEP_CHAR)) return filename + strlen(filename);
    return filename + i;
}

char *get_filename_leafname(char *filename) {
    int i = strlen(filename) - 1;
    while ((i>=0) && (filename[i] != SEP_CHAR)) i--;
    return filename + i + 1;
}

int file_exists(char *filename) {
    FILE *TEST = fopen(filename, "r");
    if (TEST) { fclose(TEST); return TRUE; }
    return FALSE;
}

long int file_size(char *filename) {
    FILE *TEST_FILE = fopen(filename, "rb");
    if (TEST_FILE) {
        if (fseek(TEST_FILE, 0, SEEK_END) == 0) {
            long int file_size = ftell(TEST_FILE);
            if (file_size == -1L) fatal_fs("ftell failed on linked file", filename);
            return file_size;
        } else fatal_fs("fseek failed on linked file", filename);
        fclose(TEST_FILE);
    }
    return -1L;
}

void copy_file(char *from, char *to) {
    if ((from == NULL) || (to == NULL) || (strcmp(from, to) == 0))
        fatal("files confused in copier");

    FILE *FROM = fopen(from, "rb"); if (FROM == NULL) fatal_fs("unable to read file", from);
    FILE *TO = fopen(to, "wb"); if (TO == NULL) fatal_fs("unable to write to file", to);

    while (TRUE) {
        int c = fgetc(FROM);
        if (c == EOF) break;
        putc(c, TO);
    }

    fclose(FROM); fclose(TO);
}
```

The function get_filename_extension is called from 2/blorb, 3/rel and 3/links.
The function get_filename_leafname is called from 1/blurb, 3/links and 3/web.
The function file_exists is called from 3/templ.
The function file_size is called from 2/blorb and 3/links.
The function copy_file is called from 3/rel.

Blurb Parser 1/blurb

Purpose

To read and follow the instructions in the blurb file, our main input.

§**1. Reading the file.** We divide the file into blurb commands at line breaks, so:

```
void parse_blurb_file(char *in) {
    file_read(in, "can't open blurb file", TRUE, interpret, 0);
    set_error_position(NULL);
}
```

The function parse_blurb_file is called from 1/main.

§**2.** The sequence of values enumerated here must correspond exactly to indexes into the syntaxes table below.

```
define author_COMMAND 0
define auxiliary_COMMAND 1
define copyright_COMMAND 2
define cover_COMMAND 3
define css_COMMAND 4
define ifiction_COMMAND 5
define ifiction_public_COMMAND 6
define ifiction_file_COMMAND 7
define palette_COMMAND 8
define palette_16_bit_COMMAND 9
define palette_32_bit_COMMAND 10
define picture_scaled_COMMAND 11
define picture_COMMAND 12
define placeholder_COMMAND 13
define project_folder_COMMAND 14
define release_COMMAND 15
define release_file_COMMAND 16
define release_file_from_COMMAND 17
define release_source_COMMAND 18
define release_to_COMMAND 19
define resolution_max_COMMAND 20
define resolution_min_max_COMMAND 21
define resolution_min_COMMAND 22
define resolution_COMMAND 23
define solution_COMMAND 24
define solution_public_COMMAND 25
define sound_music_COMMAND 26
define sound_repeat_COMMAND 27
define sound_forever_COMMAND 28
define sound_song_COMMAND 29
define sound_COMMAND 30
define source_COMMAND 31
define source_public_COMMAND 32
```

```
define storyfile_include_COMMAND 33
define storyfile_COMMAND 34
define storyfile_leafname_COMMAND 35
define template_path_COMMAND 36
define website_COMMAND 37
```

§**3.** A single number specifying various possible combinations of operands:

```
define OPS_NO 1
define OPS_1TEXT 2
define OPS_2TEXT 3
define OPS_1NUMBER 4
define OPS_2NUMBER 5
define OPS_1NUMBER_1TEXT 6
define OPS_1NUMBER_2TEXTS 7
define OPS_1NUMBER_1TEXT_1NUMBER 8
define OPS_3NUMBER 9
define OPS_3TEXT 10
```

§**4.** Each legal command syntax is stored as one of these structures. We will be parsing commands using the C library function `sscanf`, which is a little idiosyncratic. It is, in particular, not easy to find out whether `sscanf` successfully matched the whole text, since it returns only the number of variable elements matched, so that it can't tell the difference between `do %n` and `do %n quickly`, say. The text "do 12" would match against both and return 1 in each case. To get around this, we end the prototype with a spurious `" %n"`. The space can match against arbitrary white space, including none at all, and `%n` is not strictly a match – instead it sets the number of characters from the original command which have been matched. It would be nice to use `sscanf`'s return value to test whether the `%n` has been reached, but this is unsafe because the `sscanf` specification is ambiguous as to whether or not a `%n` counts towards the return value; the `man` page openly admits that people aren't sure whether it does or doesn't. So we ignore the return value of `sscanf` as meaningless, and instead test the value set by `%n` to see if it's the length of the original text.

```
typedef struct blurb_command {
    char *explicated;                    plain English form of the command
    char *prototype;                     sscanf prototype
    int operands;                        one of the above OPS_* codes
    int deprecated;
} blurb_command;
```

The structure blurb_command is private to this section.

§**5.** And here they all are. They are tested in the sequence given, and the sequence must exactly match the numbering of the `*_COMMAND` values above, since those are indexes into this table.

In blurb syntax, a line whose first non-white-space character is an exclamation mark `!` is a comment, and is ignored. (This is the I6 comment character, too.) It appears in the table as a command but, as we shall see, has no effect.

```
blurb_command syntaxes[] = {
    { "author \"name\"", "author \"%[^\"]\" %n", OPS_1TEXT, FALSE },
    { "auxiliary \"filename\" \"description\"",
            "auxiliary \"%[^\"]\" \"%[^\"]\" %n", OPS_2TEXT, FALSE },
    { "copyright \"message\"", "copyright \"%[^\"]\" %n", OPS_1TEXT, FALSE },
    { "cover \"filename\"", "cover \"%[^\"]\" %n", OPS_1TEXT, FALSE },
    { "css", "css %n", OPS_NO, FALSE },
    { "ifiction", "ifiction %n", OPS_NO, FALSE },
    { "ifiction public", "ifiction public %n", OPS_NO, FALSE },
    { "ifiction \"filename\" include", "ifiction \"%[^\"]\" include %n", OPS_1TEXT, FALSE },
    { "palette { details }", "palette {%[^}]} %n", OPS_1TEXT, TRUE },
    { "palette 16 bit", "palette 16 bit %n", OPS_NO, TRUE },
    { "palette 32 bit", "palette 32 bit %n", OPS_NO, TRUE },
    { "picture N \"filename\" scale ...",
            "picture %d \"%[^\"]\" scale %s %n", OPS_1NUMBER_2TEXTS, TRUE },
    { "picture N \"filename\"", "picture %d \"%[^\"]\" %n", OPS_1NUMBER_1TEXT, FALSE },
    { "placeholder [name] = \"text\"", "placeholder [%[A-Z]] = \"%[^\"]\" %n", OPS_2TEXT, FALSE },
    { "project folder \"pathname\"", "project folder \"%[^\"]\" %n", OPS_1TEXT, FALSE },
    { "release \"text\"", "release \"%[^\"]\" %n", OPS_1TEXT, FALSE },
    { "release file \"filename\"", "release file \"%[^\"]\" %n", OPS_1TEXT, FALSE },
    { "release file \"filename\" from \"template\"",
            "release file \"%[^\"]\" from \"%[^\"]\" %n", OPS_2TEXT, FALSE },
    { "release source \"filename\" using \"filename\" from \"template\"",
            "release source \"%[^\"]\" using \"%[^\"]\" from \"%[^\"]\" %n", OPS_3TEXT, FALSE },
    { "release to \"pathname\"", "release to \"%[^\"]\" %n", OPS_1TEXT, FALSE },
    { "resolution NxN max NxN", "resolution %d max %d %n", OPS_2NUMBER, TRUE },
    { "resolution NxN min NxN max NxN", "resolution %d min %d max %d %n", OPS_3NUMBER, TRUE },
    { "resolution NxN min NxN", "resolution %d min %d %n", OPS_2NUMBER, TRUE },
    { "resolution NxN", "resolution %d %n", OPS_1NUMBER, TRUE },
    { "solution", "solution %n", OPS_NO, FALSE },
    { "solution public", "solution public %n", OPS_NO, FALSE },
    { "sound N \"filename\" music", "sound %d \"%[^\"]\" music %n", OPS_1NUMBER_1TEXT, TRUE },
    { "sound N \"filename\" repeat N",
            "sound %d \"%[^\"]\" repeat %d %n", OPS_1NUMBER_1TEXT_1NUMBER, TRUE },
    { "sound N \"filename\" repeat forever",
            "sound %d \"%[^\"]\" repeat forever %n", OPS_1NUMBER_1TEXT, TRUE },
    { "sound N \"filename\" song", "sound %d \"%[^\"]\" song %n", OPS_1NUMBER_1TEXT, TRUE },
    { "sound N \"filename\"", "sound %d \"%[^\"]\" %n", OPS_1NUMBER_1TEXT, FALSE },
    { "source", "source %n", OPS_NO, FALSE },
    { "source public", "source public %n", OPS_NO, FALSE },
    { "storyfile \"filename\" include", "storyfile \"%[^\"]\" include %n", OPS_1TEXT, FALSE },
    { "storyfile \"filename\"", "storyfile \"%[^\"]\" %n", OPS_1TEXT, TRUE },
    { "storyfile leafname \"leafname\"", "storyfile leafname \"%[^\"]\" %n", OPS_1TEXT, FALSE },
    { "template path \"folder\"", "template path \"%[^\"]\" %n", OPS_1TEXT, FALSE },
    { "website \"template\"", "website \"%[^\"]\" %n", OPS_1TEXT, FALSE },
    { NULL, NULL, OPS_NO, FALSE }
};
```

§**6. Summary.** For the `-help` information:

```
void summarise_blurb(void) {
    int t;
    printf("\nThe blurbfile is a script of commands, one per line, in these forms:\n");
    for (t=0; syntaxes[t].prototype; t++)
        if (syntaxes[t].deprecated == FALSE)
            printf("  %s\n", syntaxes[t].explicated);
    printf("\nThe following syntaxes, though legal in Blorb 2001, are not supported:\n");
    for (t=0; syntaxes[t].prototype; t++)
        if (syntaxes[t].deprecated == TRUE)
            printf("  %s\n", syntaxes[t].explicated);
}
```

The function summarise_blurb is called from 1/main.

§**7. The interpreter.** The following routine is called for each line of the blurb file in sequence, including any blank lines.

```
void interpret(char *command, text_file_position *tf) {
    set_error_position(tf);
    if (command == NULL) fatal("null blurb line");
    command = trim_white_space(command);
    if (command[0] == 0) return;                                  thus skip a line containing only blank space
    if (command[0] == '!') return;                                thus skip a comment line
    if (trace_mode) fprintf(stdout, "! %03d: %s\n", tfp_get_line_count(tf), command);
    int outcome = -1;                                             which of the legal command syntaxes is used
    char text1[MAX_TEXT_FILE_LINE_LENGTH], text2[MAX_TEXT_FILE_LINE_LENGTH],
        text3[MAX_TEXT_FILE_LINE_LENGTH];
    text1[0] = 0; text2[0] = 0; text3[0] = 0;
    int num1 = 0, num2 = 0, num3 = 0;
    ⟨Parse the command and set operands appropriately 8⟩;
    ⟨Take action on the command 9⟩;
}
```

§**8.** Here we set `outcome` to the index in the syntaxes table of the line matched, or leave it as -1 if no match can be made. Text and number operands are copied in `text1`, `num1`, ..., accordingly.

⟨Parse the command and set operands appropriately 8⟩ ≡

```
    int t;
    for (t=0; syntaxes[t].prototype; t++) {
        char *pr = syntaxes[t].prototype;
        int nm = -1;                                              number of characters matched
        switch (syntaxes[t].operands) {
            case OPS_NO: sscanf(command, pr, &nm); break;
            case OPS_1TEXT: sscanf(command, pr, text1, &nm); break;
            case OPS_2TEXT: sscanf(command, pr, text1, text2, &nm); break;
            case OPS_1NUMBER: sscanf(command, pr, &num1, &nm); break;
            case OPS_2NUMBER: sscanf(command, pr, &num1, &num2, &nm); break;
            case OPS_1NUMBER_1TEXT: sscanf(command, pr, &num1, text1, &nm); break;
            case OPS_1NUMBER_2TEXTS: sscanf(command, pr, &num1, text1, text2, &nm); break;
            case OPS_1NUMBER_1TEXT_1NUMBER: sscanf(command, pr, &num1, text1, &num2, &nm); break;
            case OPS_3NUMBER: sscanf(command, pr, &num1, &num2, &num3, &nm); break;
```

```
            case OPS_3TEXT: sscanf(command, pr, text1, text2, text3, &nm); break;
            default: fatal("unknown operand type");
        }
        if (nm == strlen(command)) { outcome = t; break; }
    }
    if ((strlen(text1) >= MAX_FILENAME_LENGTH-1) ||
        (strlen(text2) >= MAX_FILENAME_LENGTH-1) ||
        (strlen(text3) >= MAX_FILENAME_LENGTH-1)) {
        error("string too long"); return;
    }
    if (outcome == -1) {
        error_1("not a valid blurb command", command);
        return;
    }
    if (syntaxes[outcome].deprecated) {
        error_1("this Blurb syntax is no longer supported", syntaxes[outcome].explicated);
        return;
    }
```

This code is used in §7.

§**9.** The command is now fully parsed, and is one that we support. We can act.

⟨Take action on the command 9⟩ ≡

```
    switch (outcome) {
        case author_COMMAND:
            set_placeholder_to("AUTHOR", text1, 0);
            author_chunk(text1);
            break;
        case auxiliary_COMMAND: create_auxiliary_file(text1, text2); break;
        case copyright_COMMAND: copyright_chunk(text1); break;
        case cover_COMMAND: ⟨Declare which file is the cover art 10⟩; break;
        case css_COMMAND: use_css_code_styles = TRUE; break;
        case ifiction_file_COMMAND: metadata_chunk(text1); break;
        case ifiction_COMMAND: request_1(IFICTION_REQ, "", TRUE); break;
        case ifiction_public_COMMAND: request_1(IFICTION_REQ, "", FALSE); break;
        case picture_COMMAND: picture_chunk(num1, text1); break;
        case placeholder_COMMAND: set_placeholder_to(text1, text2, 0); break;
        case project_folder_COMMAND: strcpy(project_folder, text1); break;
        case release_COMMAND:
            set_placeholder_to_number("RELEASE", num1);
            release_chunk(num1);
            break;
        case release_file_COMMAND:
            request_2(COPY_REQ, text1, get_filename_leafname(text1), FALSE); break;
        case release_file_from_COMMAND:
            request_2(RELEASE_FILE_REQ, text1, text2, FALSE); break;
        case release_to_COMMAND: strcpy(release_folder, text1); break;
        case release_source_COMMAND:
            request_3(RELEASE_SOURCE_REQ, text1, text2, text3, FALSE); break;
        case solution_COMMAND: request_1(SOLUTION_REQ, "", TRUE); break;
        case solution_public_COMMAND: request_1(SOLUTION_REQ, "", FALSE); break;
        case sound_COMMAND: sound_chunk(num1, text1); break;
```

```
        case source_COMMAND: request_1(SOURCE_REQ, "", TRUE); break;
        case source_public_COMMAND: request_1(SOURCE_REQ, "", FALSE); break;
        case storyfile_include_COMMAND: executable_chunk(text1); break;
        case storyfile_leafname_COMMAND:
            set_placeholder_to("STORYFILE", text1, 0);
            break;
        case template_path_COMMAND: new_template_path(text1); break;
        case website_COMMAND: request_1(WEBSITE_REQ, text1, FALSE); break;
        default: error_1("***", command); fatal("*** command unimplemented ***\n");
    }
```

This code is used in §7.

§**10.** We only ever set the frontispiece as resource number 1, since Inform has the assumption that the cover art is image number 1 built in.

⟨Declare which file is the cover art 10⟩ ≡

```
    set_placeholder_to("BIGCOVER", text1, 0);
    cover_exists = TRUE;
    cover_is_in_JPEG_format = TRUE;
    if ((text1[strlen(text1)-3] == 'p') || (text1[strlen(text1)-3] == 'P'))
        cover_is_in_JPEG_format = FALSE;
    frontispiece_chunk(1);
    char *leaf = get_filename_leafname(text1);
    if (cover_is_in_JPEG_format) strcpy(leaf, "Small Cover.jpg");
    else strcpy(leaf, "Small Cover.png");
    set_placeholder_to("SMALLCOVER", text1, 0);
```

This code is used in §9.

2 Blorbs

`2/blorb`: *Blorb Writer.w* To write the Blorb file, our main output, to disc.

Blorb Writer 2/blorb

Purpose

To write the Blorb file, our main output, to disc.

2/blorb.§1 Big-endian integers; §2-6 Chunks; §7-17 Our choice of chunks; §18-25 Main construction

Definitions

¶1. "Blorb" is an IF-specific format, but it is defined as a form of IFF file. IFF, "Interchange File Format", is a general-purpose wrapper format dating back to the mid-1980s; it was designed as a way to gather together audiovisual media for use on home computers. (Though Electronic Arts among others used IFF files to wrap up entertainment material, Infocom, the pioneer of IF at the time, did not.) Each IFF file consists of a chunk, but any chunk can contain other chunks in turn. Chunks are identified with initial ID texts four characters long. In different domains of computing, people use different chunks, and this makes different sorts of IFF file look like different file formats to the end user. So we have TIFF for images, AIFF for uncompressed audio, AVI for movies, GIF for bitmap graphics, and so on.

¶2. Main variables:

```
int total_size_of_Blorb_chunks = 0;              ditto, but not counting the FORM header or the RIdx chunk
int no_indexed_chunks = 0;
```

¶3. As we shall see, chunks can be used for everything from a few words of copyright text to 100MB of uncompressed choral music.

Our IFF file will consist of a front part and then the chunks, one after another, in order of their creation. Every chunk has a type, a 4-character ID like `"AUTH"` or `"JPEG"`, specifying what kind of data it holds; some chunks are also given resource", " numbers which allow the story file to refer to them as it runs – the pictures, sound effects and the story file itself all have unique resource numbers. (These are called "indexed", because references to them appear in a special `RIdx` record in the front part of the file – the "resource index".)

```
typedef struct chunk_metadata {
    char filename[MAX_FILENAME_LENGTH];                    if the content is stored on disc
    char data_in_memory[MAX_FILENAME_LENGTH];              if the content is stored in memory
    int length_of_data_in_memory;                          in bytes; or −1 if the content is stored on disc
    char *chunk_type;                                      pointer to a 4-character string
    char *index_entry;                                     ditto
    int resource_id;                                       meaningful only if this is a chunk which is indexed
    int byte_offset;                                       from the start of the chunks, which is not quite the start of the IFF file
    int size;                                              in bytes
    MEMORY_MANAGEMENT
} chunk_metadata;
```

The structure chunk_metadata is private to this section.

§**1. Big-endian integers.** IFF files use big-endian integers, whereas `cBlorb` might or might not (depending on the platform it runs on), so we need routines to write 32, 16 or 8-bit values in explicitly big-endian form:

```
void four_word(FILE *F, int n) {
    fputc((n / 0x1000000)%0x100, F);
    fputc((n / 0x10000)%0x100, F);
    fputc((n / 0x100)%0x100, F);
    fputc((n)%0x100, F);
}
void two_word(FILE *F, int n) {
    fputc((n / 0x100)%0x100, F);
    fputc((n)%0x100, F);
}
void one_byte(FILE *F, int n) {
    fputc((n)%0x100, F);
}
void s_four_word(char *F, int n) {
    F[0] = (n / 0x1000000)%0x100;
    F[1] = (n / 0x10000)%0x100;
    F[2] = (n / 0x100)%0x100;
    F[3] = (n)%0x100;
}
void s_two_word(char *F, int n) {
    F[0] = (n / 0x100)%0x100;
    F[1] = (n)%0x100;
}
void s_one_byte(char *F, int n) {
    F[0] = (n)%0x100;
}
```

§**2. Chunks.** Although chunks can be written in a nested way – that's the whole point of IFF, in fact – we will always be writing a very flat structure, in which a single enclosing chunk (`FORM`) contains a sequence of chunks with no further chunks inside.

```
chunk_metadata *current_chunk = NULL;
```

§3. Each chunk is "added" in one of two ways. *Either* we supply a filename for an existing binary file on disc which will hold the data we want to write, *or* we supply a `NULL` filename and a `data` pointer to `length` bytes in memory.

```
void add_chunk_to_blorb(char *id, int resource_num, char *supplied_filename, char *index,
    char *data, int length) {
    if (chunk_type_is_legal(id) == FALSE)
        fatal("tried to complete non-Blorb chunk");
    if (index_entry_is_legal(index) == FALSE)
        fatal("tried to include mis-indexed chunk");
    current_chunk = CREATE(chunk_metadata);
    ⟨Set the filename for the new chunk 4⟩;
    current_chunk->chunk_type = id;
    current_chunk->index_entry = index;
    if (current_chunk->index_entry) no_indexed_chunks++;
    current_chunk->byte_offset = total_size_of_Blorb_chunks;
    current_chunk->resource_id = resource_num;
    ⟨Compute the size in bytes of the chunk 5⟩;
    ⟨Advance the total chunk size 6⟩;
    if (trace_mode)
        printf("! Begun chunk %s: fn is <%s> (innate size %d)\n",
            current_chunk->chunk_type, current_chunk->filename, current_chunk->size);
}
```

§4.

⟨Set the filename for the new chunk 4⟩ ≡

```
    if (data) {
        strcpy(current_chunk->filename, "(not from a file)");
        current_chunk->length_of_data_in_memory = length;
        int i;
        for (i=0; i<length; i++) current_chunk->data_in_memory[i] = data[i];
    } else {
        strcpy(current_chunk->filename, supplied_filename);
        current_chunk->length_of_data_in_memory = -1;
    }
```

This code is used in §3.

§5.

⟨Compute the size in bytes of the chunk 5⟩ ≡

```
    int size;
    if (data) {
        size = length;
    } else {
        size = (int) file_size(supplied_filename);
    }
    if (chunk_type_is_already_an_IFF(current_chunk->chunk_type) == FALSE)
        size += 8;                                  allow 8 further bytes for the chunk header to be added later
    current_chunk->size = size;
```

This code is used in §3.

§**6.** Note the adjustment of `total_size_of_Blorb_chunks` so as to align the next chunk's position at a two-byte boundary – this betrays IFF's origin in the 16-bit world of the mid-1980s. Today's formats would likely align at four, eight or even sixteen-byte boundaries.

⟨Advance the total chunk size 6⟩ ≡

```
total_size_of_Blorb_chunks += current_chunk->size;
if ((current_chunk->size) % 2 == 1) total_size_of_Blorb_chunks++;
```

This code is used in §3.

§**7. Our choice of chunks.** We will generate only the following chunks with the above apparatus. The full Blorb specification does include others, but Inform doesn't need them.

The weasel words "with the above..." are because we will also generate two chunks separately: the compulsory `"FORM"` chunk enclosing the entire Blorb, and an indexing chunk, `"RIdx"`. Within this index, some chunks appear, but not others, and they are labelled with the "index entry" text.

```
char *legal_Blorb_chunk_types[] = {
    "AUTH", "(c) ", "Fspc", "RelN", "IFmd",          miscellaneous identifying data
    "JPEG", "PNG ",                                  images in different formats
    "AIFF", "OGGV", "MIDI", "MOD ",                  sound effects in different formats
    "ZCOD", "GLUL",                                  story files in different formats
    NULL };
char *legal_Blorb_index_entries[] = {
    "Pict", "Snd ", "Exec", NULL };
```

§**8.** Because we are wisely paranoid:

```
int chunk_type_is_legal(char *type) {
    int i;
    if (type == NULL) return FALSE;
    for (i=0; legal_Blorb_chunk_types[i]; i++)
        if (strcmp(type, legal_Blorb_chunk_types[i]) == 0)
            return TRUE;
    return FALSE;
}
int index_entry_is_legal(char *entry) {
    int i;
    if (entry == NULL) return TRUE;
    for (i=0; legal_Blorb_index_entries[i]; i++)
        if (strcmp(entry, legal_Blorb_index_entries[i]) == 0)
            return TRUE;
    return FALSE;
}
```

§**9.** Because it will make a difference to how we embed a file into our Blorb, we need to know whether the chunk in question is already an IFF in its own right. Only one type of chunk is, as it happens:

```
int chunk_type_is_already_an_IFF(char *type) {
    if (strcmp(type, "AIFF")==0) return TRUE;
    return FALSE;
}
```

§**10.** `"AUTH"`: author's name, as a null-terminated string.

```
void author_chunk(char *t) {
    if (trace_mode) printf("! Author: <%s>\n", t);
    add_chunk_to_blorb("AUTH", 0, NULL, NULL, t, strlen(t));
}
```

The function author_chunk is called from 1/blurb.

§**11.** `"(c) "`: copyright declaration.

```
void copyright_chunk(char *t) {
    if (trace_mode) printf("! Copyright declaration: <%s>\n", t);
    add_chunk_to_blorb("(c) ", 0, NULL, NULL, t, strlen(t));
}
```

The function copyright_chunk is called from 1/blurb.

§**12.** `"Fspc"`: frontispiece image ID number – which picture resource provides cover art, in other words.

```
void frontispiece_chunk(int pn) {
    if (trace_mode) printf("! Frontispiece is image %d\n", pn);
    char data[4];
    s_four_word(data, pn);
    add_chunk_to_blorb("Fspc", 0, NULL, NULL, data, 4);
}
```

The function frontispiece_chunk is called from 1/blurb.

§**13.** `"RelN"`: release number.

```
void release_chunk(int rn) {
    if (trace_mode) printf("! Release number is %d\n", rn);
    char data[2];
    s_two_word(data, rn);
    add_chunk_to_blorb("RelN", 0, NULL, NULL, data, 2);
}
```

The function release_chunk is called from 1/blurb.

§**14.** `"Pict"`: a picture, or image. This must be available as a binary file on disc, and in a format which Blorb allows: for Inform 7 use, this will always be PNG or JPEG. There can be any number of these chunks.

```
void picture_chunk(int n, char *fn) {
    char *p = get_filename_extension(fn);
    char *type = "PNG ";
    if (*p == '.') {
        p++;
        if ((*p == 'j') || (*p == 'J')) type = "JPEG";
    }
    add_chunk_to_blorb(type, n, fn, "Pict", NULL, 0);
    no_pictures_included++;
}
```

The function picture_chunk is called from 1/blurb.

§**15.** `"Snd "`: a sound effect. This must be available as a binary file on disc, and in a format which Blorb allows: for Inform 7 use, this is officially Ogg Vorbis or AIFF at present, but there has been repeated discussion about adding MOD ("SoundTracker") or MIDI files, so both are supported here.

There can be any number of these chunks, too.

```
void sound_chunk(int n, char *fn) {
    char *p = get_filename_extension(fn);
    char *type = "AIFF";
    if (*p == '.') {
        p++;
        if ((*p == 'o') || (*p == 'O')) type = "OGGV";
        else if ((*p == 'm') || (*p == 'M')) {
            if ((p[1] == 'i') || (p[1] == 'I')) type = "MIDI";
            else type = "MOD ";
        }
    }
    add_chunk_to_blorb(type, n, fn, "Snd ", NULL, 0);
    no_sounds_included++;
}
```

The function sound_chunk is called from 1/blurb.

§**16.** `"Exec"`: the executable program, which will normally be a Z-machine or Glulx story file. It's legal to make a blorb with no story file in, but Inform 7 never does this.

```
void executable_chunk(char *fn) {
    char *p = get_filename_extension(fn);
    char *type = "ZCOD";
    if (*p == '.') {
        if (p[strlen(p)-1] == 'x') type = "GLUL";
    }
    add_chunk_to_blorb(type, 0, fn, "Exec", NULL, 0);
}
```

The function executable_chunk is called from 1/blurb.

§**17.** `"IFmd"`: the bibliographic data (or "metadata") about the work of IF being blorbed up, in the form of an iFiction record. (The format of which is set out in the *Treaty of Babel* agreement.)

```
void metadata_chunk(char *fn) {
    add_chunk_to_blorb("IFmd", 0, fn, NULL, NULL, 0);
}
```

The function metadata_chunk is called from 1/blurb.

§18. Main construction.

```
void write_blorb_file(char *out) {
    if (NUMBER_CREATED(chunk_metadata) == 0) return;
    FILE *IFF = fopen(out, "wb");
    if (IFF == NULL) fatal_fs("can't open blorb file for output", out);
    int RIdx_size, first_byte_after_index;
    ⟨Calculate the sizes of the whole file and the index chunk 19⟩;
    ⟨Write the initial FORM chunk of the IFF file, and then the index 20⟩;
    if (trace_mode) ⟨Print out a copy of the chunk table 24⟩;
    chunk_metadata *chunk;
    LOOP_OVER(chunk, chunk_metadata) ⟨Write the chunk 21⟩;
    fclose(IFF);
}
```

The function write_blorb_file is called from 1/main.

§19. The bane of IFF file generation is that each chunk has to be marked up-front with an offset to skip past it. This means that, unlike with XML or other files having flexible-sized ingredients delimited by begin-end markers, we always have to know the length of a chunk before we start writing it.

That even extends to the file itself, which is a single IFF chunk of type `"FORM"`. So we need to think carefully. We will need the `FORM` header, then the header for the `RIdx` indexing chunk, then the body of that indexing chunk – with one record for each indexed chunk; and then room for all of the chunks we'll copy in, whether they are indexed or not.

⟨Calculate the sizes of the whole file and the index chunk 19⟩ ≡

```
int FORM_header_size = 12;
int RIdx_header_size = 12;
int index_entry_size = 12;
RIdx_size = RIdx_header_size + index_entry_size*no_indexed_chunks;
first_byte_after_index = FORM_header_size + RIdx_size;
blorb_file_size = first_byte_after_index + total_size_of_Blorb_chunks;
```

This code is used in §18.

§20. Each different IFF file format is supposed to provide its own "magic text" identifying what the file format is, and for Blorbs that text is "IFRS", short for "IF Resource".

⟨Write the initial FORM chunk of the IFF file, and then the index 20⟩ ≡

```
fprintf(IFF, "FORM");
four_word(IFF, blorb_file_size - 8);                    offset to end of FORM after the 8 bytes so far
fprintf(IFF, "IFRS");                                   magic text identifying the IFF as a Blorb
fprintf(IFF, "RIdx");
four_word(IFF, RIdx_size - 8);                          offset to end of RIdx after the 8 bytes so far
four_word(IFF, no_indexed_chunks);                      i.e., number of entries in the index
chunk_metadata *chunk;
LOOP_OVER(chunk, chunk_metadata)
    if (chunk->index_entry) {
        fprintf(IFF, "%s", chunk->index_entry);
        four_word(IFF, chunk->resource_id);
        four_word(IFF, first_byte_after_index + chunk->byte_offset);
    }
```

This code is used in §18.

§**21.** Most of the chunks we put in exist on disc without their headers, but AIFF sound files are an exception, because those are IFF files in their own right; so they come with ready-made headers.

⟨Write the chunk 21⟩ ≡
```
int bytes_to_copy;
char *type = chunk->chunk_type;
if (chunk_type_is_already_an_IFF(type) == FALSE) {
    fprintf(IFF, "%s", type);
    four_word(IFF, chunk->size - 8);                    offset to end of chunk after the 8 bytes so far
    bytes_to_copy = chunk->size - 8;                    since here the chunk size included 8 extra
} else {
    bytes_to_copy = chunk->size;                    whereas here the chunk size was genuinely the file size
}
if (chunk->length_of_data_in_memory >= 0)
    ⟨Copy that many bytes from memory 23⟩
else
    ⟨Copy that many bytes from the chunk file on the disc 22⟩;
if ((bytes_to_copy % 2) == 1) one_byte(IFF, 0);                                  as we allowed for above
```

This code is used in §18.

§**22.** Sometimes the chunk's contents are on disc:

⟨Copy that many bytes from the chunk file on the disc 22⟩ ≡
```
FILE *CHUNKSUB = fopen(chunk->filename, "rb");
if (CHUNKSUB == NULL) fatal_fs("unable to read data", chunk->filename);
else {
    int i;
    for (i=0; i<bytes_to_copy; i++) {
        int j = fgetc(CHUNKSUB);
        if (j == EOF) fatal_fs("chunk ran out incomplete", chunk->filename);
        one_byte(IFF, j);
    }
    fclose(CHUNKSUB);
}
```

This code is used in §21.

§**23.** And sometimes, for shorter things, they are in memory:

⟨Copy that many bytes from memory 23⟩ ≡
```
int i;
for (i=0; i<bytes_to_copy; i++) {
    int j = chunk->data_in_memory[i];
    one_byte(IFF, j);
}
```

This code is used in §21.

§**24.** For debugging purposes only:

⟨Print out a copy of the chunk table 24⟩ ≡

```
printf("! Chunk table:\n");
chunk_metadata *chunk;
LOOP_OVER(chunk, chunk_metadata)
    printf("! Chunk %s %06x %s %d <%s>\n",
        chunk->chunk_type, chunk->size,
        (chunk->index_entry)?(chunk->index_entry):"unindexed",
        chunk->resource_id,
        chunk->filename);
printf("! End of chunk table\n");
```

This code is used in §18.

3 Other Material

`3/rel`: *Releaser.w* To manage requests to release material other than a Blorb file.

`3/sol`: *Solution Deviser.w* To make a solution (.sol) file accompanying a release, if requested.

`3/links`: *Links and Auxiliary Files.w* To manage links to auxiliary files, and placeholder variables.

`3/place`: *Placeholders.w* To manage placeholder variables.

`3/templ`: *Templates.w* To manage templates for website generation.

`3/web`: *Website Maker.w* To accompany a release with a mini-website.

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

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?
    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;
    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;
}
```

§**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();
    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");
        }
    }
}
```

§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 WEBSITE_REQ: ⟨Create a website 11⟩; break;
            case SOURCE_REQ: ⟨Create a plain text source file 6⟩; break;
            case SOLUTION_REQ: ⟨Create a walkthrough file 5⟩; break;
            case IFICTION_REQ: ⟨Create an iFiction file 7⟩; break;
            case COPY_REQ: ⟨Copy a file into the release folder 8⟩; break;
            case RELEASE_FILE_REQ: ⟨Release a file into the release folder 9⟩; break;
            case RELEASE_SOURCE_REQ: ⟨Release source text as HTML into the release folder 10⟩; 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);
```

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);
```

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);
copy_file(req->details1, write_to);
```

This code is used in §4.

§9.

⟨Release a file into the release folder 9⟩ ≡
```
release_file_into_website(req->details1, req->details2);
```

This code is used in §4.

§10.

⟨Release source text as HTML into the release folder 10⟩ ≡
```
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.

§11. 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 11⟩ ≡
```
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);
    }
}
release_file_into_website("index.html", t);
request *req;
LOOP_OVER(req, request)
    if (req->private == FALSE)
        switch (req->what_is_requested) {
            case WEBSITE_REQ: break;
            case SOLUTION_REQ: break;
            case IFICTION_REQ: 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); break;
        }
    ⟨Add further material as requested by the template 12⟩;
```

This code is used in §4.

§**12.** 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 12⟩ ≡

```
    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 §11.

§**13.** If so, then `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"));
}
```

§**14.** 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 write_to[MAX_FILENAME_LENGTH];
    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 15⟩
    else
        ⟨Release a binary file from the template into the website 16⟩;
}
```

§**15.** "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 15⟩ ≡

```
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 §14.

§**16.**

⟨Release a binary file from the template into the website 16⟩ ≡

```
if (trace_mode) printf("! Binary file %s from template %s\n", name, t);
copy_file(from, write_to);
```

This code is used in §14.

§**17.** 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 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.

§**18. 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);
}
```

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); }
```

§**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⟩;
            }
        }
    }
}
```

§**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;
}
```

§**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;
}
```

§**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;
}
```

§**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]);
}
```

§**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;
}
```

§**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);
            }
        }
    }
}
```

§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⟩;
}
```

§**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);
}
```

§**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⟩;
}
```

§**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");
}
```

§**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);
}
```

Links and Auxiliary Files 3/links

Purpose

To manage links to auxiliary files, and placeholder variables.

3/links.§1 Registration; §2-3 Linking; §4-5 Links; §6 Cover image; §7 Releasing

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);
}
```

§**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");
}
```

§**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 = " 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, ", %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);
    }
}
```

§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.

3/place.§1-6 Initial values

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
    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("BLURB", "", 0);
    set_placeholder_to("TEMPLATE", "Standard", 0);
    set_placeholder_to("GENERATOR", VERSION, 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 read_placeholder is called from 1/main, 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) {
    if (strlen(text) >= MAX_FILENAME_LENGTH - 1) { error("value too long"); return; }
    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) {
        if (reservation > 0) { error("tried to set reserved variable"); return; }
        strcpy(wv->pl_contents, text);
        return;
    }
    wv = CREATE(placeholder);
    if (trace_mode) printf("! Creating [%s]\n", var);
    strcpy(wv->pl_name, var);
    strcpy(wv->pl_contents, text);
    wv->reservation = reservation;
}
```

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

§**5.** 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.)

```
void copy_placeholder_to(char *var, FILE *COPYTO) {
    int multiparagraph_mode = FALSE;
    if (strcmp(var, "BLURB") == 0) multiparagraph_mode = TRUE;
    placeholder *wv = find_placeholder(var);
    if (wv == NULL) { fprintf(COPYTO, "[%s]", var); return; }
    if (multiparagraph_mode) fprintf(COPYTO, "<p>");
    switch (wv->reservation) {
        case 0: ⟨Copy an ordinary unreserved placeholder 6⟩; 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>");
}
```

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

§**6.** 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 6⟩ ≡

```
    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] == '\x0a') || (p[i] == '\x0d') || (p[i] == '\x7f'))
            fprintf(COPYTO, "<p>");
        else fprintf(COPYTO, "%c", p[i]);
    }
```

This code is used in §5.

Templates 3/templ

Purpose

To manage templates for website generation.

3/templ.§1-4 Defining template paths; §5-6 Searching for template files

---

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;
}
```

§**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.

```
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) {
        t = CREATE(template);
        strcpy(t->template_name, name);
        t->template_location = tp;
        return t;
    }
    return NULL;
}
```

§**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).

```
char *find_file_in_named_template(char *name, char *needed) {
    template *t = find_template(name), *Standard = find_template("Standard");
    if (t == NULL) { error_1("template seems not to exist", name); return NULL; }
    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;
}
```

Website Maker 3/web

Purpose

To accompany a release with a mini-website.

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

---

Definitions

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

```
define ABBREVIATED_HEADING_LENGTH 100

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;
}
```

§**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>");
    }
}
```

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

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

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

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

§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");
    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 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");
}
```

§**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⟩;
}
```

§**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⟩;
}
```

§**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 (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⟩;
}
```

§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;
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;
    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;
    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 doc = FALSE;
    if ((segment_being_written) && (segment_being_written->documentation)) doc = TRUE;
    if (doc) fprintf(COPYTO, "%d", no_doc_files);
    else fprintf(COPYTO, "%d", no_src_files);
}
```

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 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;
    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);
}
```

§**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;
}
```

§**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)) {
    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] == '*')) {
        fprintf(SPAGE, "<p><a name=\"note%d\"></a>", next_footnote_number);
        open_style(SPAGE, "notetext");
        fprintf(SPAGE, "[%d]. ", next_footnote_number);
        next_footnote_number++;
        i+=2;
        while ((line[i]) && (line[i] != ']')) {
            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) { fprintf(SPAGE, "<tr>"); open_table_cell(SPAGE); }
int start = 0;
if (tabulate == FALSE) {
    for (; line[start] == '\t'; start++) ;
    open_code_paragraph(SPAGE, start);
}
⟨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) { close_table_cell(SPAGE); fprintf(SPAGE, "</tr>\n"); }
else close_code_paragraph(SPAGE);
```

This code is used in §35.

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

⟨Begin typographic embellishments 40⟩ ≡

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

This code is used in §39.

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

⟨End typographic embellishments 41⟩ ≡

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

This code is used in §39.

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

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

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

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

This code is used in §39.

§**43.**

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

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

This code is used in §39.

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

⟨Any heading line is in bold 44⟩ ≡

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

This code is used in §39.

§**45.**

⟨The documentation requires some corrections 45⟩ ≡
```
if ((comment_nesting == 0) && (quoted_matter == FALSE) && (i6_matter == FALSE) &&
    (line[start] == '*') && (line[start+1] == ':') && (line[start+2] == ' '))
    start += 3;
if (line_count == position_of_documentation_bar) strcpy(line, "Documentation");
```
This code is used in §39.

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

⟨Typeset a single character of the source text 46⟩ ≡
```
switch (line[i]) {
    case '\t':
        a multiple tab is equivalent to a single tab in Inform source text
        while (line[i+1] == '\t') i++;
        ⟨Typeset a tab 47⟩;
        break;
    case '"':
        if ((comment_nesting > 0) || (i6_matter)) fprintf(SPAGE, "&quot;");
        else ⟨Typeset a double quotation mark outside of a comment 48⟩;
        break;
    case '[':
        if (quoted_matter) { fprintf(SPAGE, "["); change_style(SPAGE, "substitution"); }
        else if (i6_matter) fprintf(SPAGE, "[");
        else ⟨Typeset an open square bracket outside of a string 49⟩;
        break;
    case ']':
        if (quoted_matter) { change_style(SPAGE, "quote"); fprintf(SPAGE, "]"); }
        else if (i6_matter) fprintf(SPAGE, "]");
        else ⟨Typeset a close square bracket outside of a string 50⟩;
        break;
    case '(':
        if ((comment_nesting == 0) && (quoted_matter == FALSE) && (i6_matter == FALSE) &&
            (line[i+1] == '-')) { i++;
            ⟨Typeset the opening of I6 verbatim code 51⟩
        } else fprintf(SPAGE, "("); break;
    case '-':
        if ((i6_matter) && (line[i+1] == ')')) { i++;
            ⟨Typeset the closing of I6 verbatim code 52⟩
        } else fprintf(SPAGE, "-"); break;
    case '<': fprintf(SPAGE, "&lt;"); break;
    case '>': fprintf(SPAGE, "&gt;"); break;
    case '&': fprintf(SPAGE, "&amp;"); break;
    default: fprintf(SPAGE, "%c", line[i]); break;
}
```
This code is used in §39.

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

⟨Typeset a tab 47⟩ ≡
```
if (tabulate) {
    ⟨End typographic embellishments 41⟩;
    close_table_cell(SPAGE);
    open_table_cell(SPAGE);
    ⟨Begin typographic embellishments 40⟩;
} else {
    fprintf(SPAGE, " ");
}
```

This code is used in §46.

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

Our code in handling quoted and comment matter is greatly simplified by the fact that a valid Inform text cannot contain mismatched square brackets, and nor can a valid comment contain mismatched quotation marks.

⟨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, and must not contain nested further comments.

⟨Typeset an open square bracket outside of a string 49⟩ ≡
```
if (line[i+1] == '*') {
    advance past the end of the asterisked comment
    while ((line[i]) && (line[i+1] != ']')) i++; if (line[i]) i++;
    ⟨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 `linknotes` if we have CSS, and otherwise render in grey superscript.

⟨Typeset a footnote cue 53⟩ ≡
```
open_style(SPAGE, "linknotes");
fprintf(SPAGE, "<a href=\"#note%d\">[%d]</a>",
    next_footnote_number, next_footnote_number);
close_style(SPAGE, "linknotes");
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⟩;
}
```

§**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.