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 10240                  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 51200            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 1.2"
define TRUE 1
define FALSE 0
```

¶5. The following variables record HTML and Javascript-related points where `cblorb` needs to behave differently on the different platforms. The default values here aren't actually correct for any platform as they stand: in the `main` routine below, we set them as needed.

```
char SEP_CHAR = '/';                                  local file-system filename separator
char *FONT_TAG = "size=2";                            contents of a <font> tag
char *JAVASCRIPT_PRELUDE = "javascript:window.Project."; calling prefix
int escape_openUrl = FALSE, escape_fileUrl = FALSE;
int reverse_slash_openUrl = FALSE, reverse_slash_fileUrl = FALSE;
```

¶6. Some global variables:

```
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 HTML_pages_created = 0;                      number of pages created in the website, if any
int source_HTML_pages_created = 0;               number of those holding source

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
char status_template[MAX_FILENAME_LENGTH];       filename of report HTML page template, if any
char status_file[MAX_FILENAME_LENGTH];           filename of report HTML page to write, if any
int cover_exists = FALSE;                        an image is specified as cover art
int default_cover_used = FALSE;                  but it's only the default supplied by Inform
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 6⟩; return 0; }
    parse_blurb_file(blurb_filename);
    write_blorb_file(blorb_filename);
    create_requested_material();
    print_report();
    free_memory();
    if (error_count > 0) return 1;
    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;
    status_file[0] = 0;
    status_template[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;
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;
    }
    ⟨Parse an individual command-line argument 4⟩;
}
⟨Set platform-dependent HTML and Javascript variables 5⟩;
if (project_folder[0] != 0) {
    if (names > 0) ⟨Command line syntax error 7⟩;
    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.

⟨Parse an individual command-line argument 4⟩ ≡

```
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 7⟩;
    strcpy(project_folder, argv[arg]);
    continue;
}
if (p[0] == '-') ⟨Command line syntax error 7⟩;
names++;
switch (names) {
    case 1: strcpy(blurb_filename, p); break;
    case 2: strcpy(blorb_filename, p); break;
    default: ⟨Command line syntax error 7⟩;
}
```

This code is used in §3.

§**5.** Now let's set the platform-dependent variables – all of which depend only on the value of `platform`.

`cblorb` generates quite a variety of HTML, for instance to create websites, but the tricky points below affect only one special page not browsed by the general public: the results page usually called `StatusCblorb.html` (though this depends on how the `status` command is used in the blurb). The results page is intended only for viewing within the Inform user interface, and it expects to have two Javascript functions available, `openUrl` and `fileUrl`. Because the object structure has needed to be different for the Windows and OS X user interface implementations of Javascript, we abstract the prefix for these function calls into the `JAVASCRIPT_PRELUDE`. Thus

```
<a href="***openUrl">...</a>
```

causes a link, when clicked, to call the `openUrl` function, where `***` is the prelude; similarly for `fileUrl`. The first opens a URL in the local operating system's default web browser, the second opens a file (identified by a `file:...` URL) in the local operating system. These two URLs may need treatment to handle special characters:

(a) "escaping", where spaces in the URL are escaped to `%2520`, which within a Javascript string literal produces `%20`, the standard way to represent a space in a web URL;

(b) "reversing slashes", where backslashes are converted to forward slashes – useful if the separation character is a backslash, as on Windows, since backslashes are escape characters in Javascript literals.

⟨Set platform-dependent HTML and Javascript variables 5⟩ ≡

```
if (platform == OSX_PLATFORM) {
    FONT_TAG = "face=\"lucida grande,geneva,arial,tahoma,verdana,helvetica,helv\" size=2";
    escape_openUrl = TRUE;                    OS X requires openUrl to escape, and fileUrl not to
}
if (platform == WINDOWS_PLATFORM) {
    SEP_CHAR = '\\';
    JAVASCRIPT_PRELUDE = "javascript:external.Project.";
    reverse_slash_openUrl = TRUE; reverse_slash_fileUrl = TRUE;
}
```

This code is used in §3.

§**6.**

⟨Produce help 6⟩ ≡

```
printf("This is cblorb, a component of Inform 7 for packaging up IF materials.\n\n");
⟨Show command line usage 8⟩;
summarise_blurb();
```

This code is used in §1.

§**7.**

⟨Command line syntax error 7⟩ ≡

```
⟨Show command line usage 8⟩;
return 1;
```

This code is used in §3,4.

§**8.**

⟨Show command line usage 8⟩ ≡

```
    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 §6,7.

§**9. 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);
}
```

The function establish_time is.

§**10.** 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.

§**11. 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");
}
```

The function print_banner is.

§**12.** The concluding banner is much smaller – empty if all went well, a single comment line if not. But we also generate the status report page (if that has been requested) – a single HTML file generated from a template by expanding placeholders in the template. All of the meat of the report is in those placeholders, of course; the template contains only some fancy formatting.

```
void print_report(void) {
    if (error_count > 0) printf("! Completed: %d error(s)\n", error_count);
    ⟨Set a whole pile of placeholders which will be needed to generate the status page 13⟩;
    if (status_template[0]) web_copy(status_template, status_file);
}
```

The function print_report is called from 1/text.

§**13.** If it isn't apparent what these placeholders do, take a look at the template file for `StatusCblorb.html` in the Inform application – that's where they're used.

⟨Set a whole pile of placeholders which will be needed to generate the status page 13⟩ ≡

```
    if (error_count > 0) {
        set_placeholder_to("CBLORBSTATUS", "Failed", 0);
        set_placeholder_to("CBLORBSTATUSIMAGE", "inform:/cblorb_failed.png", 0);
        set_placeholder_to("CBLORBSTATUSTEXT",
            "Inform translated your source text as usual, to manufacture a 'story "
            "file': all of that worked fine. But the Release then went wrong, for "
            "the following reason:<p><ul>[CBLORBERRORS]</ul>", 0
        );
    } else {
        set_placeholder_to("CBLORBERRORS", "No problems occurred", 0);
        set_placeholder_to("CBLORBSTATUS", "Succeeded", 0);
        set_placeholder_to("CBLORBSTATUSIMAGE", "file://[SMALLCOVER]", 0);
        set_placeholder_to("CBLORBSTATUSTEXT",
            "All went well. I've put the released material into the 'Release' subfolder "
            "of the Materials folder for the project: you can take a look with "
            "the menu option <b>Release &gt; Open Materials Folder</b> or by clicking "
            "the blue folders above.<p>"
            "Releases can range in size from a single blorb file to a medium-sized website. "
            "Here's what we currently have:<p>", 0
        );
        report_requested_material("CBLORBSTATUSTEXT");
    }
    if (blorb_file_size > 0) {
        set_placeholder_to_number("BLORBFILESIZE", blorb_file_size/1024);
```

```
        set_placeholder_to_number("BLORBFILEPICTURES", no_pictures_included);
        set_placeholder_to_number("BLORBFILESOUNDS", no_sounds_included);
        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 {
        set_placeholder_to_number("BLORBFILESIZE", 0);
        set_placeholder_to_number("BLORBFILEPICTURES", 0);
        set_placeholder_to_number("BLORBFILESOUNDS", 0);
        printf("! Completed: no blorb output requested\n");
    }
```

This code is used in §12.