CBLORB

The Program

Chapter 1

Build 3/100422   Graham Nelson

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

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 §3,4,3,4.

§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 §6,7,6,7.

§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.

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⟩;
}
```

The function allocate_another_block is.

§**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;
        }
}
```

The function check_memory_integrity is.

The function debug_memory_frames is.

§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;
}
```

The function allocate_mem is.

§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

Purpose

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

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(char *t, text_file_position *tfp) {
    *t = 0;
    if (tfp == NULL) return;
    sprintf(t, "%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) {
    char err[MAX_FILENAME_LENGTH];
    describe_file_position(err, error_position);
    sprintf(err+strlen(err), "Error: %s\n", erm);
    spool_error(err);
}

void error_1(char *erm, char *s) {
    char err[MAX_FILENAME_LENGTH];
    describe_file_position(err, error_position);
    sprintf(err+strlen(err), "Error: %s: '%s'\n", erm, s);
    spool_error(err);
}

void errorf_1s(char *erm, char *s1) {
    char err[MAX_FILENAME_LENGTH];
    sprintf(err, erm, s1);
    spool_error(err);
}

void errorf_2s(char *erm, char *s1, char *s2) {
    char err[MAX_FILENAME_LENGTH];
    sprintf(err, erm, s1, s2);
    spool_error(err);
}

void fatal(char *erm) {
    char err[MAX_FILENAME_LENGTH];
    describe_file_position(err, error_position);
    sprintf(err+strlen(err), "Fatal error: %s\n", erm);
    spool_error(err);
    print_report();
    exit(1);
}

void fatal_fs(char *erm, char *fn) {
    char err[MAX_FILENAME_LENGTH];
    describe_file_position(err, error_position);
    sprintf(err+strlen(err), "Fatal error: %s: filename '%s'\n", erm, fn);
    spool_error(err);
    print_report();
    exit(1);
}

void warning_fs(char *erm, char *fn) {
    char err[MAX_FILENAME_LENGTH];
    describe_file_position(err, error_position);
    fprintf(stderr, "%sWarning: %s: filename '%s'\n", err, erm, fn);
}
```

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 and 3/web.
The function errorf_1s is called from 3/rel and 3/templ.
The function errorf_2s is called from 3/rel.
The function fatal is called from 1/mem, 1/blurb, 2/blorb and 3/web.
The function fatal_fs is called from 2/blorb, 3/sol and 3/b64.
The function warning_fs is.

§5. Errors are spooled to a placeholder, for the benefit of the report:

```
void spool_error(char *err) {
    append_to_placeholder("CBLORBERRORS", "<li>");
    append_to_placeholder("CBLORBERRORS", err);
    append_to_placeholder("CBLORBERRORS", "</li>");
    fprintf(stderr, "%s", err);
    error_count++;
}
```

The function spool_error is.

§6. 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 7⟩;
    ⟨Set the initial position, seeking it in the file if need be 8⟩;
    ⟨Read in lines and send them one by one to the iterator 9⟩;
    fclose(HANDLE);
}
```

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

§7.

```
⟨Open the text file 7⟩ ≡
    if (strlen(filename) >= MAX_FILENAME_LENGTH) {
        if (serious) fatal_fs("filename too long", filename);
        error_1("filename too long", filename);
        return;
    }
    HANDLE = fopen(filename, "rb");
    if (HANDLE == NULL) {
        if (message == NULL) return;
        if (serious) fatal_fs(message, filename);
        else { error_1(message, filename); return; }
    }
```

This code is used in §6.

§8. 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 8⟩ ≡

```
    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 §6.

§9. 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 9⟩ ≡

```
    char line[MAX_TEXT_FILE_LINE_LENGTH+1];
    int i = 0, c = ' ';
    int warned = FALSE;
    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 10⟩;
                if (c == '\x0a') tfp.skip_terminator = '\x0d';
                if (c == '\x0d') tfp.skip_terminator = '\x0a';
            } else tfp.skip_terminator = 'X';
            ⟨Update the text file position 11⟩;
            i = 0;
        } else {
            if (i < MAX_TEXT_FILE_LINE_LENGTH) line[i++] = (char) c;
            else {
                if (serious) fatal_fs("line too long", filename);
                if (warned == FALSE) {
                    warning_fs("line too long (truncating it)", filename);
                    warned = TRUE;
                }
            }
        }
    }
    if ((i > 0) && (tfp.actively_scanning))
        ⟨Feed the completed line to the iterator routine 10⟩;
```

This code is used in §6.

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

⟨Feed the completed line to the iterator routine 10⟩ ≡

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

This code is used in §9.

§11. 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 11⟩ ≡

```
    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 §9.

§12. 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.

§13.

```
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.

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

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

The function white_space is.

§**15. 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);
            fclose(TEST_FILE);
            return file_size;
        } else fatal_fs("fseek failed on linked file", filename);
        fclose(TEST_FILE);
    }
    return -1L;
}
int copy_file(char *from, char *to, int suppress_error) {
    if ((from == NULL) || (to == NULL) || (strcmp(from, to) == 0))
        fatal("files confused in copier");
    FILE *FROM = fopen(from, "rb");
    if (FROM == NULL) {
        if (suppress_error == FALSE) fatal_fs("unable to read file", from);
        return -1;
    }
    FILE *TO = fopen(to, "wb");
    if (TO == NULL) {
        fatal_fs("unable to write to file", to);
        return -1;
    }
    int size = 0;
    while (TRUE) {
        int c = fgetc(FROM);
        if (c == EOF) break;
        size++;
        putc(c, TO);
    }
    fclose(FROM); fclose(TO);
    return size;
}
```

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

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 `base64_COMMAND 2`
define `copyright_COMMAND 3`
define `cover_COMMAND 4`
define `css_COMMAND 5`
define `ifiction_COMMAND 6`
define `ifiction_public_COMMAND 7`
define `ifiction_file_COMMAND 8`
define `interpreter_COMMAND 9`
define `palette_COMMAND 10`
define `palette_16_bit_COMMAND 11`
define `palette_32_bit_COMMAND 12`
define `picture_scaled_COMMAND 13`
define `picture_COMMAND 14`
define `placeholder_COMMAND 15`
define `project_folder_COMMAND 16`
define `release_COMMAND 17`
define `release_file_COMMAND 18`
define `release_file_from_COMMAND 19`
define `release_source_COMMAND 20`
define `release_to_COMMAND 21`
define `resolution_max_COMMAND 22`
define `resolution_min_max_COMMAND 23`
define `resolution_min_COMMAND 24`
define `resolution_COMMAND 25`
define `solution_COMMAND 26`
define `solution_public_COMMAND 27`
define `sound_music_COMMAND 28`
define `sound_repeat_COMMAND 29`
define `sound_forever_COMMAND 30`
define `sound_song_COMMAND 31`
define `sound_COMMAND 32`

define `source_COMMAND 33`
define `source_public_COMMAND 34`
define `status_COMMAND 35`
define `status_alternative_COMMAND 36`
define `status_instruction_COMMAND 37`
define `storyfile_include_COMMAND 38`
define `storyfile_COMMAND 39`
define `storyfile_leafname_COMMAND 40`
define `template_path_COMMAND 41`
define `website_COMMAND 42`

§**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 },
    { "base64 \"filename\" to \"filename\"",
            "base64 \"%[^\"]\" to \"%[^\"]\" %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 },
    { "interpreter \"interpreter-name\" \"vm-letter\"",
        "interpreter \"%[^\"]\" \"%[gz]\" %n", OPS_2TEXT, 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 },
    { "status \"template\" \"filename\"", "status \"%[^\"]\" \"%[^\"]\" %n", OPS_2TEXT, FALSE },
    { "status alternative ||link to Inform documentation||",
        "status alternative ||%[^|]|| %n", OPS_1TEXT, FALSE },
    { "status instruction ||link to Inform source text||",
        "status instruction ||%[^|]|| %n", OPS_1TEXT, 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⟩;
}
```

The function interpret is.

§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 base64_COMMAND:
            request_2(BASE64_REQ, text1, text2, FALSE); 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 interpreter_COMMAND:
            set_placeholder_to("INTERPRETERVMIS", text2, 0);
            request_1(INTERPRETER_REQ, text1, 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);
            ⟨Make pathname placeholders in three different formats 11⟩;
            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 status_COMMAND: strcpy(status_template, text1); strcpy(status_file, text2); break;
        case status_alternative_COMMAND: request_1(ALTERNATIVE_REQ, text1, FALSE); break;
        case status_instruction_COMMAND: request_1(INSTRUCTION_REQ, text1, 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 (strcmp(leaf, "DefaultCover.jpg") == 0) default_cover_used = TRUE;
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.

§11. Here, `text1` is the pathname of the Release folder. If we suppose that `cblorb` is being run from Inform, then this folder is a subfolder of the Materials folder for an I7 project. It follows that we can obtain the pathname to the Materials folder by trimming the leaf and the final separator. That makes the `MATERIALSFOLDERPATH` placeholder. We then set `MATERIALSFOLDER` to the name of the Materials folder, e.g., "Spaceman Spiff Materials".

However, we also need two variants on the pathname, one to be supplied to the Javascript function `openUrl` and one to `fileUrl`. For platform dependency reasons these need to be manipulated to deal with awkward characters.

⟨Make pathname placeholders in three different formats 11⟩ ≡

```
set_placeholder_to("MATERIALSFOLDERPATH", text1, 0);
int k = strlen(text1);
while ((k>=0) && (text1[k] != SEP_CHAR)) k--;
if (k>0) { *(read_placeholder("MATERIALSFOLDERPATH")+k)=0; k--; }
while ((k>=0) && (text1[k] != SEP_CHAR)) k--; k++;
set_placeholder_to("MATERIALSFOLDER", text1 + k, 0);
char *L = read_placeholder("MATERIALSFOLDER");
while (*L) { if (*L == SEP_CHAR) *L = 0; L++; }
qualify_placeholder("MATERIALSFOLDERPATHOPEN", "MATERIALSFOLDERPATHFILE",
    "MATERIALSFOLDERPATH");
```

This code is used in §9.

§12. And here that very "qualification" routine. The placeholder `original` contains the pathname to a folder, a pathname which might contain spaces or backslashes, and which needs to be quoted as a literal Javascript string supplied to either the function `openUrl` or the function `fileUrl`. Depending on the platform in use, this may entail escaping spaces or reversing slashes in the pathname in order to make versions for these two functions to use.

```
void qualify_placeholder(char *openUrl_path, char *fileUrl_path, char *original) {
    int i;
    char *p = read_placeholder(original);
    for (i=0; p[i]; i++) {
        char oU_glyph[8], fU_glyph[8];
        sprintf(oU_glyph, "%c", p[i]); sprintf(fU_glyph, "%c", p[i]);
        if (p[i] == ' ') {
            if (escape_openUrl) sprintf(oU_glyph, "%%2520");
            if (escape_fileUrl) sprintf(fU_glyph, "%%2520");
        }
        if (p[i] == '\\') {
            if (reverse_slash_openUrl) sprintf(oU_glyph, "/");
            if (reverse_slash_fileUrl) sprintf(fU_glyph, "/");
        }
        append_to_placeholder(openUrl_path, oU_glyph);
        append_to_placeholder(fileUrl_path, fU_glyph);
    }
}
```

The function qualify_placeholder is.