Text Files 1/text

Purpose

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

1/text.§1-3 Text file positions; §4-5 Error messages; §6-11 File handling; §12-14 Two string utilities; §15 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(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.