The `CTIE` processor

(Version 1.1 [TEX Live])

March 17, 2021 at 13:08

1*   Introduction.   Whenever a programmer wants to change a given `WEB` or `CWEB` program (referred to as a `WEB` program throughout this program) because of system dependencies, she or he will create a new change file. In addition there may be a second change file to modify system independent modules of the program. But the `WEB` file cannot be tangled and weaved with more than one change file simultaneously. The `TIE` program was designed to merge a `WEB` file and several change files producing a new `WEB` file, and since the input files are tied together, the program was called `TIE`. Furthermore, the program could be used to merge several change files giving a new single change file. This method seems to be more important because it doesn't modify the original source file.

However, the introduction of `CWEB` has meant that `TIE` is not quite able to perform its task correctly any longer: `CWEB` introduced the idea of include files, which are input into `CWEB` files using the `@i` command, and `TIE` is unable to handle such constructs if the change files modify lines included in those files. The present program, `CTIE`, is designed to overcome this lack. Like `TIE`, upon which it is based, it can either output a single master `WEB` file or a master change file. However, in both cases, any include commands will be totally expanded and the files included in the output rather than the `@i` commands being left; this makes this code feasible, which it would not necessarily be otherwise. Other than this difference, `CTIE` should function identically to `TIE` on files which do not involve any `CWEB` include commands.

The algorithm used is essentially the same as that of `TIE`, with modifications to check for and handle `@i` commands. Thus, as with `TIE`, the method used only needs one buffer line for each input file. Thus the storage requirement of `CTIE` does not depend on the sizes of the input files but only on their number.

The program is written in C and has few system dependencies.

The "banner line" defined here should be changed whenever `CTIE` is modified. We also keep the version number here separately for ease; it is used below.

#define *version_number* `"1.1"`
#define *banner* `"This␣is␣CTIE,␣Version␣1.1"`
#define *copyright* `"Copyright␣2002,2003␣Julian␣Gilbey."`
                    `"␣␣All␣rights␣reserved.␣␣There␣is␣no␣warranty.\n"`
                    `"Run␣with␣the␣--version␣option␣for␣other␣important␣information."`

3*   And this is the structure of the *main* function: this is where `CTIE` starts, and where it ends.

⟨ The main function 3* ⟩ ≡
  int *main*(**int** *argc*, **string** ∗*argv*)
  {
    ⟨ Set up `PROGNAME` feature and initialise the search path mechanism 71* ⟩
    ⟨ Initialise parameters 17 ⟩
    ⟨ Scan the parameters 61 ⟩
    ⟨ Print the banners 60* ⟩
    ⟨ Get the master file started 40* ⟩
    ⟨ Prepare the change files 41* ⟩
    ⟨ Prepare the output file 38* ⟩
    ⟨ Process the input 57 ⟩
    ⟨ Check that all changes have been read 58 ⟩
    exit(*wrap_up*( ));
  }
This code is used in section 2.

4*   We include the additional types **boolean** and **string**. `CTIE` replaces the complex `TIE` character set handling (based on that of the original `WEB` system) with the standard `CWEB` behaviour, and so uses the **char** type for input and output.

The `kpathsea` library (version 3.4.5 and higher) defines the **boolean** (with the values *true* and *false*) and **string** (and **const_string**) types in `<kpathsea/simpletypes.h>`, so we do not actually need to define them here.

5* We don't need to predeclare any string handling functions here, as the `kpathsea` headers do the right thing by including `<string.h>` behind the scenes.

6* The following parameters should be sufficient for most applications of CTIE.

#**define** buf_size 1024   ▷ maximum length of one input line ◁
#**define** max_file_index 32   ▷ we don't think that anyone needs more than 32 change files ◁
#**define** $xisupper(c)$ $(isupper((\textbf{unsigned char})\ c) \wedge ((\textbf{unsigned char})\ c < °\mathit{200}))$

7* We introduce a history variable that allows us to set a return code if the operating system can use it. First we introduce the coded values for the history. This variable must be initialised. (We do this even if the value given may be the default for variables, just to document the need for the initial value.)

⟨ Global variables 7* ⟩ ≡
  typedef enum {
    $spotless, troublesome, fatal$
  } **return_code**;
  static return_code $history \leftarrow spotless$;

See also sections 15*, 16, 18, 22, 39*, and 66*.

This code is used in section 2.

8.* Input and output.   Standard output for the user is done by writing on *stdout*. Error messages
are written to *stderr*. Terminal input is not needed in this version of CTIE. *stdin*, *stdout* and *stderr* are
predefined as we include the <stdio.h> definitions through the kpathsea interface.

⟨ Global **#include**s 8* ⟩ ≡
#**include** <kpathsea/kpathsea.h>

This code is used in section 2.

9.*   And we need dynamic memory allocation. This should cause no trouble in any C program. The
kpathsea include files handle the definition of *malloc*, too.

10*   Data structures.   The multiple primary input files (master file and change files) are treated the same way. To organise the simultaneous usage of several input files, we introduce the data type **in_file_modes**.

The mode *search* indicates that CTIE searches for a match of the input line with any line of an input file in *reading* mode. *test* is used whenever a match is found and it has to be tested if the next input lines do match also. *reading* describes that the lines can be read without any check for matching other lines. *ignore* denotes that the file cannot be used. This may happen because an error has been detected or because the end of the file has been found.

file_types is used to describe whether a file is a master file or a change file. The value *unknown* is added to this type to set an initial mode for the output file. This enables us to check whether any option was used to select the kind of output. (This would even be necessary if we would assume a default action for missing options.)

⟨ Global types 10* ⟩ ≡
  typedef enum {
    search, *test*, *reading*, *ignore*
  } **in_file_modes**;
  typedef enum {
    unknown, *master*, *chf*
  } **file_types**;
See also sections 11*, 12, 13, and 14.

This code is used in section 2.

11*   A variable of type **out_md_type** will tell us in what state the output change file is during processing. *normal* will be the state, when we did not yet start a change, *pre* will be set when we write the lines to be changes and *post* will indicate that the replacement lines are written.

⟨ Global types 10* ⟩ +≡
  typedef enum {
    normal, *pre*, *post*
  } **out_md_type**;

15*   Every one of the primary input files might include another file using the @i include mechanism. In turn, each of these might include other files, and so on. We allow a limited number of these files to be opened simultaneously, and we store information about the currently open include files as a linked list attached to each primary file.

#define *max_include_files* 20   ▷ maximum number of include files open simultaneously ◁
#define *max_file_name_length* 1024

⟨ Global variables 7* ⟩ +≡
  int *total_include_files* ← 0;   ▷ count 'em ◁

19*   File I/O.   The basic function *get_line* can be used to get a line from an input file. The line is stored in the *buffer* part of the descriptor. The components *limit* and *line* are updated. If the end of the file is reached *mode* is set to *ignore*. On some systems it might be useful to replace tab characters by a proper number of spaces since several editors used to create change files insert tab characters into a source file not under control of the user. So it might be a problem to create a matching change file.

We define *get_line* to read a line from a file specified by the corresponding file descriptor. This function returns *true* if it is successful and *false* if the end of the file has been reached.

⟨ Internal functions 19* ⟩ ≡
```
  static boolean get_line(file_index i, boolean do_includes)
  {
    register input_description *inp_desc ← input_organisation[i];
    register FILE *fp;

    if (inp_desc→mode ≡ ignore) return false;
  restart:
    if (inp_desc→current_include ≠ Λ) {
      register include_description *inc_desc ← inp_desc→current_include;

      fp ← inc_desc→the_file;  ⟨ Get include line into buffer or goto restart if end of file 24* ⟩
    }
    else {
      fp ← inp_desc→the_file;  ⟨ Get line into buffer, return false if end of file 20* ⟩
    }
    if (do_includes) ⟨ Check for @i in newly read line, goto restart if include fails 26 ⟩
    return true;
  }
```
See also sections 32*, 42*, 43*, 46*, 47*, 48*, and 59*.

This code is used in section 2.

20* Lines must fit into the buffer completely. We read all characters sequentially until an end of line is found (but do not forget to check for EOF!). Too long input lines will be truncated. This will result in a damaged output if they occur in the replacement part of a change file, or in an incomplete check if the matching part is concerned. Tab character expansion might be done here.

⟨ Get line into buffer, return false if end of file 20* ⟩ ≡
```
  {
    register int c;      ▷ the actual character read ◁
    register char *k;      ▷ where the next character goes ◁

    if (feof(fp)) ⟨ Handle end of file and return 21 ⟩
    inp_desc→limit ← k ← inp_desc→buffer;        ▷ beginning of buffer ◁
    while (k ≤ inp_desc→buffer_end ∧ (c ← getc(fp)) ≠ EOF ∧ c ≠ '\n')
      if ((*(k++) ← c) ≠ '␣' ∧ c ≠ '\r') inp_desc→limit ← k;
    if (k > inp_desc→buffer_end)
      if ((c ← getc(fp)) ≠ EOF ∧ c ≠ '\n') {
        ungetc(c, fp);  inp_desc→loc ← inp_desc→buffer;  err_print(i, "!␣Input␣line␣too␣long");
      }
    if (c ≡ EOF ∧ inp_desc→limit ≡ inp_desc→buffer) ⟨ Handle end of file and return 21 ⟩
    ⟨ Increment the line number and print a progress report at certain times 23 ⟩
  }
```
This code is used in section 19*.

24* The following is very similar to the above, but for the case where we are reading from an include file.

⟨ Get include line into buffer or **goto** *restart* if end of file 24* ⟩ ≡

```
  {
    register int c;        ▷ the actual character read ◁
    register char *k;        ▷ where the next character goes ◁
    if (feof (fp)) ⟨Handle end of include file and goto restart 25⟩
    inp_desc→limit ← k ← inp_desc→buffer;        ▷ beginning of buffer ◁
    while (k ≤ inp_desc→buffer_end ∧ (c ← getc(fp)) ≠ EOF ∧ c ≠ '\n')
      if ((*(k++) ← c) ≠ '␣' ∧ c ≠ '\r') inp_desc→limit ← k;
    if (k > inp_desc→buffer_end)
      if ((c ← getc(fp)) ≠ EOF ∧ c ≠ '\n') {
        ungetc(c, fp); inp_desc→loc ← inp_desc→buffer; err_print(i, "!␣Input␣line␣too␣long");
      }
    if (c ≡ EOF ∧ inp_desc→limit ≡ inp_desc→buffer) ⟨Handle end of include file and goto restart 25⟩
    inc_desc→line ++;
  }
```

This code is used in section 19*.

27* When an @i line is found in the file, we must temporarily stop reading it and start reading from the named include file. The @i line should give a complete file name with or without double quotes. We use the KPATHSEA library (in particular, the CWEBINPUTS variable) to search for this file. The remainder of the @i line after the file name is ignored.

#define *too_long*()
     {
        total_include_files −−; *free*(*new_inc*); *err_print*(*i*, "!␣Include␣file␣name␣too␣long");
        goto *restart*;
     }

⟨ Try to open include file, abort push if unsuccessful, go to *restart* 27* ⟩ ≡
  {
    include_description ∗*new_inc*;
    char ∗*file_name_end*;
    string *fullname*;
    char ∗*k*;

    new_inc ← (**include_description** ∗) *malloc*(**sizeof**(**include_description**));
    if (*new_inc* ≡ Λ) *fatal_error*(*i*, "!␣No␣memory␣for␣new␣include␣descriptor", "");
    new_inc→*line* ← 0; *k* ← *new_inc*→*file_name*; *file_name_end* ← *k* + *max_file_name_length* − 1;
    if (∗*inp_desc*→*loc* ≡ ʼ"ʼ) {
      inp_desc→*loc*++;
      while (∗*inp_desc*→*loc* ≠ ʼ"ʼ ∧ *k* ≤ *file_name_end*) ∗*k*++ ← ∗*inp_desc*→*loc*++;
      if (*inp_desc*→*loc* ≡ *inp_desc*→*limit*) *k* ← *file_name_end* + 1;   ▷ unmatched quote is 'too long' ◁
    }
    else
      while (∗*inp_desc*→*loc* ≠ ʼ␣ʼ ∧ ∗*inp_desc*→*loc* ≠ ʼ\tʼ ∧ ∗*inp_desc*→*loc* ≠ ʼ"ʼ ∧ *k* ≤ *file_name_end*)
        ∗*k*++ ← ∗*inp_desc*→*loc*++;
    if (*k* > *file_name_end*) *too_long*();
    ∗*k* ← ʼ\0ʼ; *fullname* ← *kpse_find_cweb*(*new_inc*→*file_name*);
    if (*fullname*) *new_inc*→*the_file* ← *fopen*(*fullname*, "r");
    if (*fullname* ≠ Λ ∧ *new_inc*→*the_file* ≠ Λ) {
      free(*fullname*); *new_inc*→*parent* ← *inp_desc*→*current_include*;   ▷ link it in ◁
      inp_desc→*current_include* ← *new_inc*; **goto** *restart*;   ▷ success ◁
    }
    total_include_files −−; *free*(*new_inc*);
    if (*fullname*) {
      free(*fullname*); *err_print*(*i*, "!␣Cannot␣open␣include␣file");
    }
    else *err_print*(*i*, "!␣Cannot␣find␣include␣file");
    goto *restart*;
  }

This code is used in section 26.

28*   Reporting errors to the user.   There may be errors if a line in a given change file does not match a line in the master file or a replacement in a previous change file. Such errors are reported to the user by saying

$$err\_print(\mathit{file\_no}, \texttt{"!$_\sqcup$Error$_\sqcup$message"});$$

where $\mathit{file\_no}$ is the number of the file which is concerned by the error. Please note that no trailing dot is supplied in the error message because it is appended by $\mathit{err\_print}$.

⟨ Predeclaration of functions 28* ⟩ ≡
  void err_print(**file_index**, **const char** ∗);

See also sections 33*, 35*, and 67*.

This code is used in section 2.

29*   Here is the outline of the $\mathit{err\_print}$ function.

⟨ Error handling functions 29* ⟩ ≡
  void err_print(**file_index** i, **const char** ∗s)   ▷ prints '.' and location of error message ◁
  {
    char ∗k, ∗l;   ▷ pointers into an appropriate buffer ◁
    $fprintf(stderr, *s \equiv \texttt{'!'} ? \texttt{"\textbackslash n\%s"} : \texttt{"\%s"}, s)$;
    if $(i \geq 0)$ ⟨ Print error location based on input buffer 30 ⟩
    else $putc(\texttt{'\textbackslash n'}, stderr)$;
    $fflush(stderr)$; $history \leftarrow troublesome$;
  }

See also section 36*.

This code is used in section 2.

32*   Some implementations may wish to pass the $history$ value to the operating system so that it can be used to govern whether or not other programs are started. Here, for instance, we pass the operating system a status of 0 if and only if only harmless messages were printed.

⟨ Internal functions 19* ⟩ +≡
  int $wrap\_up(\textbf{void})$
  {
    ⟨ Print the job $history$ 34 ⟩;
    if $(history > spotless)$ **return** EXIT_FAILURE;
    else return EXIT_SUCCESS;
  }

33*   Always good to prototype.

⟨ Predeclaration of functions 28* ⟩ +≡
  int $wrap\_up(\textbf{void})$;

35*   If there's a system error, we may be able to give the user more information with the $\mathit{pfatal\_error}$ function. This prints out system error information if it is available.

⟨ Predeclaration of functions 28* ⟩ +≡
  void $pfatal\_error$(**const char** ∗, **const char** ∗);

36* ⟨Error handling functions 29*⟩ +≡
  void *pfatal_error*(**const char** ∗*s*, **const char** ∗*t*)
  {
    char ∗*strerr* ← *strerror*(*errno*);
    fprintf(*stderr*, "\n%s%s", *s*, *t*);
    if (*strerr*) *fprintf*(*stderr*, "␣(%s)\n", *strerr*);
    else *fprintf*(*stderr*, "\n");
    history ← *fatal*; *exit*(*wrap_up*());
  }

37* The <errno.h> include file for the above comes via the kpathsea interface.

38* Handling multiple change files.   In the standard version we take the name of the files from the command line. It is assumed that filenames can be used as given in the command line without changes.

First there are some sections to open all files. If a file is not accessible, the run will be aborted. Otherwise the name of the open file will be displayed.

⟨ Prepare the output file 38* ⟩ ≡
  {
    out_file ← *fopen*(*out_name*, "wb");
    if (*out_file* ≡ Λ) {
      pfatal_error("!␣Cannot␣open/create␣output␣file", "");
    }
  }

This code is used in section 3*.

39*   The name of the file and the file descriptor are stored in global variables.

⟨ Global variables 7* ⟩ +≡
  FILE *out_file;
  string *out_name*;

40*   For the master file we start by reading its first line into the buffer, if we could open it. We use the kpathsea library to find the file.

⟨ Get the master file started 40* ⟩ ≡
  {
    string *fullname*;

    fullname ← *kpse_find_cweb*(*input_organisation*[0]→*file_name*);
    if (*fullname*) *input_organisation*[0]→*the_file* ← *fopen*(*fullname*, "r");
    if (*fullname* ≡ Λ ∨ *input_organisation*[0]→*the_file* ≡ Λ) {
      if (*fullname*) {
        pfatal_error("!␣Cannot␣open␣master␣file␣", *input_organisation*[0]→*file_name*);
      }
      else {
        fatal_error(−1, "!␣Cannot␣find␣master␣file␣", *input_organisation*[0]→*file_name*);
      }
    }
    else *free*(*fullname*);
    printf("(%s)\n", *input_organisation*[0]→*file_name*); *input_organisation*[0]→*type_of_file* ← *master*;
    get_line(0, *true*);
  }

This code is used in section 3*.

41* For the change files we must skip any comment part and see whether there are any changes in it. This is done by *init_change_file*.

⟨ Prepare the change files 41* ⟩ ≡

```
{
   file_index i;
   string fullname;

   i ← 1;
   while (i < no_ch) {
      fullname ← kpse_find_cweb(input_organisation[i]→file_name);
      if (fullname) input_organisation[i]→the_file ← fopen(fullname, "r");
      if (fullname ≡ Λ ∨ input_organisation[i]→the_file ≡ Λ) {
         if (fullname) {
            pfatal_error("!␣Cannot␣open␣change␣file␣", input_organisation[i]→file_name);
         }
         else {
            fatal_error(−1, "!␣Cannot␣find␣change␣file␣", input_organisation[i]→file_name);
         }
      }
      else free(fullname);
      printf("(%s)\n", input_organisation[i]→file_name);  init_change_file(i);  i++;
   }
}
```

This code is used in section 3*.

42* Input/output organisation.   Here's a simple function that checks if two lines are different.

⟨ Internal functions 19* ⟩ +≡
  static boolean *lines_dont_match*(**file_index** *i*, **file_index** *j*)
  {
    register input_description *∗iptr ← input_organisation*[*i*], *∗jptr ← input_organisation*[*j*];
    if (*iptr⃗limit − iptr⃗buffer ≠ jptr⃗limit − jptr⃗buffer*) **return** *true*;
    return *strncmp*(*iptr⃗buffer*, *jptr⃗buffer*, *iptr⃗limit − iptr⃗buffer*);
  }

43*   Function *init_change_file*(*i*) is used to ignore all lines of the input file with index *i* until the next change
module is found.

⟨ Internal functions 19* ⟩ +≡
  static void *init_change_file*(**file_index** *i*)
  {
    register input_description *∗inp_desc ← input_organisation*[*i*];
    char *ccode*;
    inp_desc⃗limit ← inp_desc⃗buffer; ⟨ Skip over comment lines; **return** if end of file 44* ⟩
    ⟨ Skip to the next nonblank line; **return** if end of file 45 ⟩
    inp_desc⃗dont_match ← 0;
  }

44*   While looking for a line that begins with @x in the change file, we allow lines that begin with @, as
long as they don't begin with @y, @z or @i (which would probably mean that the change file is fouled up).

⟨ Skip over comment lines; **return** if end of file 44* ⟩ ≡
  while (1) {
    if (¬*get_line*(*i*, *false*)) **return**;   ▷ end of file reached ◁
    if (*inp_desc⃗limit < inp_desc⃗buffer + 2*) **continue**;
    if (*inp_desc⃗buffer*[0] ≠ '@') **continue**;
    ccode ← inp_desc⃗buffer[1];
    if (*xisupper*(*ccode*)) *ccode ← tolower*((**unsigned char**) *ccode*);
    if (*ccode ≡* 'x') **break**;
    if (*ccode ≡* 'y' ∨ *ccode ≡* 'z' ∨ *ccode ≡* 'i') {
      inp_desc⃗loc ← inp_desc⃗buffer + 2; *err_print*(*i*, "!␣Missing␣@x␣in␣change␣file");
    }
  }
This code is used in section 43*.

46*   The *put_line* function is used to write a line from input buffer *j* to the output file.

⟨ Internal functions 19* ⟩ +≡
  static void *put_line*(**file_index** *j*)
  {
    char *∗ptr ← input_organisation*[*j*]*⃗buffer*;
    char *∗lmt ← input_organisation*[*j*]*⃗limit*;
    while (*ptr < lmt*) *putc*(*∗ptr++*, *out_file*);
    putc('\n', *out_file*);
  }

47* The function $e\_of\_ch\_module$ returns true if the input line from file i starts with @z.

⟨ Internal functions 19* ⟩ +≡
```
  static boolean e_of_ch_module (file_index i)
  {
    register input_description *inp_desc ← input_organisation[i];
    if (inp_desc→limit ≡ Λ) {
       err_print(i, "!␣Change␣file␣ended␣without␣@z"); return true;
    }
    else if (inp_desc→limit ≥ inp_desc→buffer + 2)
       if (inp_desc→buffer[0] ≡ '@' ∧ (inp_desc→buffer[1] ≡ 'Z' ∨ inp_desc→buffer[1] ≡ 'z')) return true;
    return false;
  }
```

48* The function $e\_of\_ch\_preamble$ returns *true* if the input line from file i starts with @y.

⟨ Internal functions 19* ⟩ +≡
```
  static boolean e_of_ch_preamble (file_index i)
  {
    register input_description *inp_desc ← input_organisation[i];
    if (inp_desc→limit ≥ inp_desc→buffer + 2 ∧ inp_desc→buffer[0] ≡ '@')
       if (inp_desc→buffer[1] ≡ 'Y' ∨ inp_desc→buffer[1] ≡ 'y') {
          if (inp_desc→dont_match > 0) {
             inp_desc→loc ← inp_desc→buffer + 2; fprintf (stderr, "\n!␣Hmm...␣%d␣", inp_desc→dont_match);
             err_print(i, "of␣the␣preceding␣lines␣failed␣to␣match");
          }
          return true;
       }
    return false;
  }
```

59* We want to tell the user about our command line options if they made a mistake. This is done by the $usage\_error()$ function. It contains merely the necessary print statements and exits afterwards.

⟨ Internal functions 19* ⟩ +≡
```
  static void usage_error (void)
  {
    ⟨ Print the banners 60* ⟩;
    fprintf (stderr, "Usage:␣ctie␣-m|-c␣outfile␣master␣changefile(s)\n");
    fprintf (stderr, "Type␣ctie␣--help␣for␣more␣information\n"); exit(EXIT_FAILURE);
  }
```

60* Printing our welcome banners; we only do this if we are not asked for version or help information.

⟨ Print the banners 60* ⟩ ≡
```
  printf ("%s␣(%s)\n", banner, kpathsea_version_string);    ▷ print a "banner line" ◁
  printf ("%s\n", copyright);     ▷ include the copyright notice ◁
```
This code is used in sections 3* and 59*.

63* We have to distinguish whether this is the very first file name (which is the case if $no\_ch \equiv none$) or if the next element of *input_organisation* must be filled.

⟨ Get a file name 63* ⟩ ≡
  {
    if ($no\_ch \equiv none$) {
      $out\_name \leftarrow *argv$;
    }
    else {
      register input_description *inp_desc;

      $inp\_desc \leftarrow$ (**input_description** *) $malloc$(**sizeof**(**input_description**));
      if ($inp\_desc \equiv \Lambda$) $fatal\_error$(-1, "!␣No␣memory␣for␣input␣descriptor", "");
      $inp\_desc\rightarrow mode \leftarrow search$; $inp\_desc\rightarrow line \leftarrow 0$; $inp\_desc\rightarrow type\_of\_file \leftarrow chf$;
      $inp\_desc\rightarrow limit \leftarrow inp\_desc\rightarrow buffer$; $inp\_desc\rightarrow buffer[0] \leftarrow$ '␣'; $inp\_desc\rightarrow loc \leftarrow inp\_desc\rightarrow buffer + 1$;
      $inp\_desc\rightarrow buffer\_end \leftarrow inp\_desc\rightarrow buffer + buf\_size - 2$; $inp\_desc\rightarrow file\_name \leftarrow *argv$;
      $inp\_desc\rightarrow current\_include \leftarrow \Lambda$; $input\_organisation[no\_ch] \leftarrow inp\_desc$;
    }
    $no\_ch ++$;
  }

This code is used in section 61.

66* Here is the usage information for `--help`.

⟨ Global variables 7* ⟩ +≡
  const_string CTIEHELP[] ← {"Usage:␣ctie␣−m|−c␣outfile␣master␣changefile(s)",
      "␣␣Create␣a␣new␣master␣file␣or␣change␣file␣from␣the␣given",
      "␣␣master␣(C)WEB␣file␣and␣changefiles.",
      "␣␣All␣filenames␣are␣taken␣literally;␣no␣suffixes␣are␣added.", "",
      "−m␣␣create␣a␣new␣master␣file␣from␣original␣(C)WEB␣and␣change␣file(s)",
      "−c␣␣␣create␣a␣master␣change␣file␣for␣original␣(C)WEB␣file␣from␣changefile(s)",
      "−−help␣␣␣␣␣␣␣display␣this␣help␣and␣exit",
      "−−version␣␣␣display␣version␣information␣and␣exit", Λ};

67* ⟨ Predeclaration of functions 28* ⟩ +≡
  static void $usage\_help$(**void**);
  static void $print\_version\_and\_exit$(**const_string**, **const_string**);

68* **static void** $usage\_help$(**void**)
  {
    const_string *$message \leftarrow$ CTIEHELP;
    while (*$message$) {
      $fputs$(*$message$, $stdout$); $putchar$('\n'); ++$message$;
    }
    $putchar$('\n'); $exit$(EXIT_SUCCESS);
  }

69*   static void *print_version_and_exit*(**const_string** *name*, **const_string** *version*)
{
  printf("%s␣%s\n", *name*, *version*); *puts*(*kpathsea_version_string*);
  puts("Copyright␣(C)␣2002,2003␣Julian␣Gilbey.");
  puts("Kpathsea␣is␣copyright␣(C)␣1999␣Free␣Software␣Foundation,␣Inc.");
  puts("There␣is␣NO␣warranty.␣␣This␣is␣free␣software.");
  puts("Redistribution␣of␣this␣software␣is␣covered␣by␣the␣terms␣of");
  puts("both␣the␣CTIE␣copyright␣and␣the␣GNU␣General␣Public␣Licence.");
  puts("For␣more␣information␣about␣these␣matters,␣see␣the␣files");
  puts("named␣COPYING␣and␣the␣CTIE␣source.");
  puts("Primary␣author␣of␣CTIE:␣Julian␣Gilbey.");
  puts("Kpathsea␣written␣by␣Karl␣Berry␣and␣others."); *exit*(EXIT_SUCCESS);
}

70* System-dependent changes.   The `ctie` program from the original `CTIE` package uses the compile-time default directory or the value of the environment variable `CWEBINPUTS` as an alternative place to be searched for files, if they could not be found in the current directory.

This version uses the KPATHSEA mechanism for searching files. The directories to be searched for come from three sources:

> (a) a user-set environment variable `CWEBINPUTS` (overridden by `CWEBINPUTS_ctie`);
> (b) a line in KPATHSEA configuration file `texmf.cnf`,
>   e.g., `CWEBINPUTS=$TEXMFDOTDIR:$TEXMF/texmf/cweb//`
>   or `CWEBINPUTS.ctie=$TEXMFDOTDIR:$TEXMF/texmf/cweb//`;
> (c) compile-time default directories (specified in `texmf.in`),
>   i.e., `$TEXMFDOTDIR:$TEXMF/texmf/cweb//`.

#define $kpse\_find\_cweb(name)$ $kpse\_find\_file(name, kpse\_cweb\_format, true)$

71*   The simple file searching is replaced by the 'path searching' mechanism that the KPATHSEA library provides.

We set $kpse\_program\_name$ to '`ctie`'. This means if the variable `CWEBINPUTS.ctie` is present in `texmf.cnf` (or `CWEBINPUTS_ctie` in the environment) its value will be used as the search path for filenames. This allows different flavors of `CTIE` to have different search paths.

⟨ Set up `PROGNAME` feature and initialise the search path mechanism 71* ⟩ ≡
  $kpse\_set\_program\_name(argv[0], \texttt{"ctie"});$

This code is used in section 3*.

72* Index.

The following sections were changed by the change file: 1, 3, 4, 5, 6, 7, 8, 9, 10, 11, 15, 19, 20, 24, 27, 28, 29, 32, 33, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 46, 47, 48, 59, 60, 63, 66, 67, 68, 69, 70, 71, 72.