The CTIE processor

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.\
            \nRun␣with␣the␣--version␣option␣for␣other␣important␣information."
```

2. The main outline of the program is now given. This can be used more or less for any C program.

```
⟨ Global #includes 8 ⟩
⟨ Global types 4 ⟩
⟨ Predeclaration of functions 5 ⟩
⟨ Global variables 7 ⟩
⟨ Error handling functions 29 ⟩
⟨ Internal functions 19 ⟩
⟨ The main function 3 ⟩
```

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

```
⟨ The main function 3 ⟩ ≡
  main(argc, argv)
      int argc;
  string * argv;
  {
    ⟨ 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.

```
#define false 0
#define true 1
⟨ Global types 4 ⟩ ≡
  typedef int boolean;
  typedef char *string;
```

See also sections 10, 11, 12, 13, and 14.

This code is used in section 2.

5. We predeclare some standard string-handling functions here instead of including their system header files, because the names of the header files are not as standard as the names of the functions. (There's confusion between `<string.h>` and `<strings.h>`.)

```
⟨ Predeclaration of functions 5 ⟩ ≡
  extern int strlen( );      ▷ length of string ◁
  extern char *strcpy( );      ▷ copy one string to another ◁
  extern int strncmp( );      ▷ compare up to n string characters ◁
  extern char *strncpy( );      ▷ copy up to n string characters ◁
  extern char *strerror( );
```

See also sections 28, 33, 35, and 67.

This code is used in section 2.

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(c) ∧ ((unsigned char) c < °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 initialized. (We do this even if the value given may be the default for variables, just to document the need for the initial value.)

```
#define spotless 0
#define troublesome 1
#define fatal 2
⟨ Global variables 7 ⟩ ≡
  int history ← 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.

⟨ Global **#include**s 8 ⟩ ≡
```
#include <stdio.h>
```
See also sections 9 and 37.

This code is used in section 2.

9. And we need dynamic memory allocation. This should cause no trouble in any C program.

⟨ Global **#include**s 8 ⟩ +≡
```
#ifdef __STDC__
#include <stdlib.h>
#else
#include <malloc.h>
#endif
```

10. Data structures. The multiple primary input files (master file and change files) are treated the same way. To organize 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 4 ⟩ +≡
#define search 0
#define test 1
#define reading 2
#define ignore 3
  typedef int in_file_modes;      ▷ should be enum (search, test, reading, ignore) ◁
#define unknown 0
#define master 1
#define chf 2
  typedef int file_types;      ▷ should be enum (unknown, master, chf) ◁
```

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 4 ⟩ +≡
#define normal 0
#define pre 1
#define post 2
  typedef int out_md_type;      ▷ should be enum (normal, pre, post) ◁
```

12. The next type will indicate variables used as an index into the file table.

```
⟨ Global types 4 ⟩ +≡
  typedef int file_index;      ▷ −1..max_file_index + 1 ◁
```

13. This is the data structure in which we collect information about each include file.

```
⟨ Global types 4 ⟩ +≡
  typedef struct _indsc {
    char file_name[max_file_name_length];
    long line;
    FILE *the_file;
    struct _indsc *parent;
  } include_description;
```

14. The following data structure contains all of the information needed to use these input files.

format *line dummy*

```
⟨ Global types 4 ⟩ +≡
  typedef struct _idsc {
    string file_name;
    char buffer[buf_size];
    in_file_modes mode;
    long line;
    file_types type_of_file;
    include_description *current_include;
    char *buffer_end;
    char *limit;
    char *loc;
    FILE *the_file;
    int dont_match;
  } input_description;
```

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 60
⟨ Global variables 7 ⟩ +≡
  int total_include_files ← 0;      ▷ count 'em ◁
```

16. The following variables refer to the files in action, the number of change files, the mode of operation and the current output state.

```
⟨ Global variables 7 ⟩ +≡
  file_index actual_input, test_input, no_ch;
  file_types prod_chf ← unknown;
  out_md_type out_mode;
```

17. And the *actual_input* and *out_mode* variables need to be initialised sensibly.

```
⟨ Initialise parameters 17 ⟩ ≡
  actual_input ← 0; out_mode ← normal;
```

This code is used in section 3.

18. All primary input files (including the master file) are recorded in the following structure. The components are usually accessed through a local pointer variable, requiring only a one-time-computation of the index expression.

```
⟨ Global variables 7 ⟩ +≡
  input_description *input_organisation[max_file_index + 1];
```

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 ⟩ ≡
  boolean get_line(i, do_includes)
      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 incomplerte 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) ≠ '␣') 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 progess report at certain times 23 ⟩
  }
```

This code is used in section 19.

21. End of file is special if this file is the master file. Then we set the global flag variable *input_has_ended*.

```
⟨ Handle end of file and return 21 ⟩ ≡
  {
    inp_desc→mode ← ignore; inp_desc→limit ← Λ;      ▷ mark end-of-file ◁
    if (inp_desc→type_of_file ≡ master) input_has_ended ← true;
    fclose(fp); return false;
  }
```

This code is used in section 20.

22. This variable must be declared for global access.

```
⟨ Global variables 7 ⟩ +≡
  boolean input_has_ended ← false;
```

23. This section does what its name says. Every 100 lines in the master file we print a dot, every 500 lines the number of lines is shown.

```
⟨ Increment the line number and print a progess report at certain times 23 ⟩ ≡
  inp_desc→line ++;
  if (inp_desc→type_of_file ≡ master ∧ inp_desc→line % 100 ≡ 0) {
    if (inp_desc→line % 500 ≡ 0) printf("%ld", inp_desc→line);
    else putchar('.');
    fflush(stdout);
  }
```

This code is used in section 20.

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) ≠ '␣') 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.

25. We don't bail out if we find the end of an include file, we just return to the parent file.

```
⟨ Handle end of include file and goto restart 25 ⟩ ≡
  {
    include_description *temp ← inc_desc→parent;
    fclose(fp); free(inc_desc); total_include_files −−; inp_desc→current_include ← temp; goto restart;
  }
```

This code is used in section 24.

26. Usually, we have to check the line we have just read to see whether it begins with `@i` and therefore needs expanding.

```
⟨ Check for @i in newly read line, goto restart if include fails 26 ⟩ ≡
  {
    inp_desc→loc ← inp_desc→buffer; *inp_desc→limit ← '␣';
    if (*inp_desc→buffer ≡ '@' ∧ (inp_desc→buffer[1] ≡ 'i' ∨ inp_desc→buffer[1] ≡ 'I')) {
      inp_desc→loc ← inp_desc→buffer + 2; *inp_desc→limit ← '"';
        ▷ this will terminate the search in all cases ◁
      while (*inp_desc→loc ≡ '␣' ∨ *inp_desc→loc ≡ '\t') inp_desc→loc++;
      if (inp_desc→loc ≥ inp_desc→limit) {
        err_print(i, "!␣Include␣file␣name␣not␣given"); goto restart;
      }
      if (total_include_files ≥ max_include_files) {
        err_print(i, "!␣Too␣many␣nested␣includes"); goto restart;
      }
      total_include_files++;      ▷ push input stack ◁
      ⟨ Try to open include file, abort push if unsuccessful, go to restart 27 ⟩;
    }
  }
```

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. If the environment variable `CWEBINPUTS` is set, or if the compiler flag of the same name was defined at compile time, `CWEB` will look for include files in the directory thus named, if it cannot find them in the current directory. (Colon-separated paths are not supported.) 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 temp_file_name[max_file_name_length];
    char *file_name_end;
    char *k, *kk;
    int l;      ▷ length of file name ◁
    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 ≡ '"') {
      inp_desc→loc++;
      while (*inp_desc→loc ≠ '"' ∧ 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 ≠ ' ' ∧ *inp_desc→loc ≠ '\t' ∧ *inp_desc→loc ≠ '"' ∧ k ≤ file_name_end)
        *k++ ← *inp_desc→loc++;
    if (k > file_name_end) too_long();
    *k ← '\0';
    if ((new_inc→the_file ← fopen(new_inc→file_name, "r")) ≠ Λ) {
      new_inc→parent ← inp_desc→current_include;      ▷ link it in ◁
      inp_desc→current_include ← new_inc; goto restart;      ▷ success ◁
    }
    kk ← getenv("CWEBINPUTS");
    if (kk ≠ Λ) {
      if ((l ← strlen(kk)) > max_file_name_length − 2) too_long();
      strcpy(temp_file_name, kk);
    }
    else {
#ifdef CWEBINPUTS
      if ((l ← strlen(CWEBINPUTS)) > max_file_name_length − 2) too_long();
      strcpy(temp_file_name, CWEBINPUTS);
#else
      l ← 0;
#endif      ▷ CWEBINPUTS ◁
    }
    if (l > 0) {
      if (k + l + 2 ≥ file_name_end) too_long();
      for ( ; k ≥ new_inc→file_name; k--) *(k + l + 1) ← *k;
```

```
    strcpy(new_inc→file_name, temp_file_name); new_inc→file_name[l] ← '/';
      ▷ UNIX pathname separator ◁
    if ((new_inc→the_file ← fopen(new_inc→file_name, "r")) ≠ Λ) {
      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); err_print(i, "! Cannot open 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

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

where *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 *err_print*.

```
⟨ Predeclaration of functions 5 ⟩ +≡
  void err_print( );
```

29. Here is the outline of the *err_print* function.

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

See also section 36.

This code is used in section 2.

30. The error locations can be indicated by using the variables *loc*, *line* and *file_name* within the appropriate file description structures, which tell respectively the first unlooked-at position in the *buffer*, the current line number and the current file. We can determine whether we are looking at an included file or not by examining the *current_include* variable. This routine should be modified on systems whose standard text editor has special line-numbering conventions.

```
⟨ Print error location based on input buffer 30 ⟩ ≡
  {
    register input_description *inp_desc ← input_organisation[i];
    register include_description *inc_desc ← inp_desc→current_include;
    if (inc_desc ≠ Λ) {
      fprintf(stderr, ". (l. %ld of include file %s", inc_desc→line, inc_desc→file_name);
      fprintf(stderr, " included from l. %ld of %s file %s)\n", inp_desc→line,
          inp_desc→type_of_file ≡ master ? "master" : "change", inp_desc→file_name);
    }
    else fprintf(stderr, ". (l. %ld of %s file %s)\n", inp_desc→line,
          inp_desc→type_of_file ≡ master ? "master" : "change", inp_desc→file_name);
    l ← (inp_desc→loc ≥ inp_desc→limit ? inp_desc→limit : inp_desc→loc);
    if (l > inp_desc→buffer) {
      for (k ← inp_desc→buffer; k < l; k++)
        if (*k ≡ '\t') putc(' ', stderr);
        else putc(*k, stderr);      ▷ print the characters already read ◁
      putc('\n', stderr);
      for (k ← inp_desc→buffer; k < l; k++) putc(' ', stderr);      ▷ space out the next line ◁
    }
    for (k ← l; k < inp_desc→limit; k++) putc(*k, stderr);      ▷ print the part not yet read ◁
    putc('\n', stderr);
  }
```

This code is used in section 29.

31. Non recoverable errors are handled by calling *fatal_error* that outputs a message and then calls '*wrap_up*' and exits. *err_print* will print the error message followed by an indication of where the error was spotted in the source files. *fatal_error* cannot state any files because the problem is usually to access these.

```
#define fatal_error(i, s, t)
        {
          fprintf(stderr, "\n%s", s); err_print(i, t); history ← fatal; exit(wrap_up());
        }
```

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()
  {
    ⟨ Print the job history 34 ⟩;
    if (history > spotless) return 1;
    else return 0;
  }
```

33. Always good to prototype.

```
⟨ Predeclaration of functions 5 ⟩ +≡
  int wrap_up( );
```

34. We report the history to the user, although this may not be "UNIX" style—but we are in good company: WEB and TEX do the same. We put this on *stdout* rather than *stderr*, so that users can easily filter this away if they wish.

```
⟨ Print the job history 34 ⟩ ≡
  switch (history) {
  case spotless: printf("\n(No␣errors␣were␣found.)\n"); break;
  case troublesome: printf("\n(Pardon␣me,␣but␣I␣think␣I␣spotted␣something␣wrong.)\n"); break;
  case fatal: printf("(That␣was␣a␣fatal␣error,␣my␣friend.)\n");
  }      ▷ there are no other cases ◁
```

This code is used in section 32.

35. If there's a system error, we may be able to give the user more information with the *pfatal_error* function. This prints out system error information if it is available.

```
⟨ Predeclaration of functions 5 ⟩ +≡
  void pfatal_error( );
```

36.

```
⟨ Error handling functions 29 ⟩ +≡
  void pfatal_error(s, t)
       char *s, *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. We need an include file for the above.

```
⟨ Global #includes 8 ⟩ +≡
#include <errno.h>
```

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, "w");
    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 desciptor 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.

```
⟨ Get the master file started 40 ⟩ ≡
  {
    input_organisation[0]→the_file ← fopen(input_organisation[0]→file_name, "r");
    if (input_organisation[0]→the_file ≡ Λ)
      pfatal_error("! Cannot open master file ", input_organisation[0]→file_name);
    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;
    i ← 1;
    while (i < no_ch) {
      input_organisation[i]→the_file ← fopen(input_organisation[i]→file_name, "r");
      if (input_organisation[i]→the_file ≡ Λ)
        pfatal_error("! Cannot open change file ", input_organisation[i]→file_name);
      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 ⟩ +≡
  boolean lines_dont_match(i, j)
      file_index i, j;
  {
    register input_description *iptr ← input_organisation[i], *jptr ← input_organisation[j];
    if (iptr→limit − iptr→buffer ≠ jptr→limit − jptr→buffer) return true;
    return strncmp(iptr→buffer, jptr→buffer, iptr→limit − iptr→buffer);
  }
```

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 ⟩ +≡
  void init_change_file(i)
      file_index i;
  {
    register input_description *inp_desc ← input_organisation[i];
    char ccode;
    inp_desc→limit ← inp_desc→buffer; ⟨ Skip over comment lines; return if end of file 44 ⟩
    ⟨ Skip to the next nonblank line; return if end of file 45 ⟩
    inp_desc→dont_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_desc→limit < inp_desc→buffer + 2) continue;
    if (inp_desc→buffer[0] ≠ '@') continue;
    ccode ← inp_desc→buffer[1];
    if (xisupper(ccode)) ccode ← tolower(ccode);
    if (ccode ≡ 'x') break;
    if (ccode ≡ 'y' ∨ ccode ≡ 'z' ∨ ccode ≡ 'i') {
      inp_desc→loc ← inp_desc→buffer + 2; err_print(i, "! Missing @x in change file");
    }
  }
```

This code is used in section 43.

45. Here we are looking at lines following the `@x`.

```
⟨ Skip to the next nonblank line; return if end of file 45 ⟩ ≡
  do {
    if (¬get_line(i, true)) {
      err_print(i, "! Change file ended after @x"); return;
    }
  } while (inp_desc→limit ≡ inp_desc→buffer);
```

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 ⟩ +≡
  void put_line(j)
      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 ⟩ +≡
  boolean e_of_ch_module(i)
      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 ⟩ +≡
  boolean e_of_ch_preamble(i)
      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;
  }
```

49. To process the input file the next section reads a line of the current (actual) input file and updates the *input_organisation* for all files with index greater than *actual_input*.

```
⟨ Process a line, break when end of source reached 49 ⟩ ≡
  {
    file_index test_file;
    ⟨ Check the current files for any ends of changes 50 ⟩
    if (input_has_ended ∧ actual_input ≡ 0) break;      ▷ all done ◁
    ⟨ Scan all other files for changes to be done 51 ⟩
    ⟨ Handle output 52 ⟩
    ⟨ Step to next line 56 ⟩
  }
```

This code is used in section 57.

50. Any of the current change files may have reached the end of the current change. In such a case, intermediate lines must be skipped and the next start of change is to be found. This may make a change file become inactive if the end of the file is reached.

```
⟨ Check the current files for any ends of changes 50 ⟩ ≡
  {
    register input_description *inp_desc;
    while (actual_input > 0 ∧ e_of_ch_module(actual_input)) {
      inp_desc ← input_organisation[actual_input];
      if (inp_desc→type_of_file ≡ master) {      ▷ emergency exit, everything mixed up! ◁
        fatal_error(−1, "!␣This␣can't␣happen:␣change␣file␣is␣master␣file", "");
      }
      inp_desc→mode ← search; init_change_file(actual_input);
      while ((input_organisation[actual_input]→mode ≠ reading ∧ actual_input > 0)) actual_input−−;
    }
  }
```

This code is used in section 49.

51. Now we will set *test_input* to the first change file that is being tested against the current line. If no other file is testing, then *actual_input* refers to a line to write and *test_input* is set to *none*.

```
#define none (-1)
⟨ Scan all other files for changes to be done 51 ⟩ ≡
  test_input ← none; test_file ← actual_input;
  while (test_input ≡ none ∧ test_file < no_ch − 1) {
    test_file++;
    switch (input_organisation[test_file]→mode) {
    case search:
      if (lines_dont_match(actual_input, test_file) ≡ false) {
        input_organisation[test_file]→mode ← test; test_input ← test_file;
      }
      break;
    case test:
      if (lines_dont_match(actual_input, test_file)) {
          ▷ error, sections do not match; just note at this point ◁
        input_organisation[test_file]→dont_match++;
      }
      test_input ← test_file; break;
    case reading:     ▷ this can't happen ◁
      break;
    case ignore:      ▷ nothing to do ◁
      break;
    }
  }
```

This code is used in section 49.

52. For the output we must distinguish between whether we are creating a new change file or a new master file. Change file creation requires closer inspection because we may be before a change, in the pattern (match) part or in the replacement part. For master file creation, we simply have to write the line from the current (actual) input.

```
⟨ Handle output 52 ⟩ ≡
  if (prod_chf ≡ chf) {
    while (1) {
      ⟨ Test for normal, break when done 53 ⟩
      ⟨ Test for pre, break when done 54 ⟩
      ⟨ Test for post, break when done 55 ⟩
    }
  }
  else if (test_input ≡ none) put_line(actual_input);
```

This code is used in section 49.

53. Check whether we have to start a change file entry. Without a match nothing needs to be done.

```
⟨ Test for normal, break when done 53 ⟩ ≡
  if (out_mode ≡ normal) {
    if (test_input ≠ none) {
      fprintf(out_file, "@x\n"); out_mode ← pre;
    }
    else break;
  }
```

This code is used in section 52.

54. Check whether we have to start the replacement text. This is the case when we are in *pre* mode but have no more matching lines. Otherwise the master file source line must be copied to the change file.

```
⟨ Test for pre, break when done 54 ⟩ ≡
  if (out_mode ≡ pre) {
    if (test_input ≡ none) {
      fprintf(out_file, "@y\n"); out_mode ← post;
    }
    else {
      if (input_organisation[actual_input]→type_of_file ≡ master) put_line(actual_input);
      break;
    }
  }
```

This code is used in section 52.

55. Check whether an entry from a change file is complete. If the current input is from a change file which is not being tested against a later change file, then this change file line must be written. If the actual input has been reset to the master file, we can finish this change.

```
⟨ Test for post, break when done 55 ⟩ ≡
  if (out_mode ≡ post) {
    if (input_organisation[actual_input]→type_of_file ≡ chf) {
      if (test_input ≡ none) put_line(actual_input);
      break;
    }
    else {
      fprintf(out_file, "@z\n\n"); out_mode ← normal;
    }
  }
```

This code is used in section 52.

56. If we had a change, we must proceed in the actual file to be changed and in the change file in effect.

```
⟨ Step to next line 56 ⟩ ≡
  get_line(actual_input, true);
  if (test_input ≠ none) {
    get_line(test_input, true);
    if (e_of_ch_preamble(test_input) ≡ true) {
      get_line(test_input, true);      ▷ update current changing file ◁
      input_organisation[test_input]→mode ← reading; actual_input ← test_input; test_input ← none;
    }
  }
```

This code is used in section 49.

57. To create the new output file we have to scan the whole master file and all changes in effect when it ends. At the very end it is wise to check for all changes to have completed, in case the last line of the master file was to be changed.

```
⟨ Process the input 57 ⟩ ≡
  actual_input ← 0; input_has_ended ← false;
  while (input_has_ended ≡ false ∨ actual_input ≠ 0)
    ⟨ Process a line, break when end of source reached 49 ⟩
  if (out_mode ≡ post)      ▷ last line has been changed ◁
    fprintf(out_file, "@z\n");
```

This code is used in section 3.

58. At the end of the program, we will tell the user if the change file had a line that didn't match any relevant line in the master file or any of the change files.

```
⟨ Check that all changes have been read 58 ⟩ ≡
  {
    file_index i;
    for (i ← 1; i < no_ch; i++) {      ▷ all change files ◁
      if (input_organisation[i]→mode ≠ ignore) {
        input_organisation[i]→loc ← input_organisation[i]→buffer;
        err_print(i, "! Change file entry did not match");
      }
    }
  }
```

This code is used in section 3.

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 ⟩ +≡
  void usage_error( )
  {
    ⟨ Print the banners 60 ⟩;
    fprintf(stderr, "Usage: ctie [-mc] outfile master changefile(s)\n");
    fprintf(stderr, "Type ctie --help for more information\n"); exit(1);
  }
```

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\n", banner);      ▷ print a "banner line" ◁
  printf("%s\n", copyright);      ▷ include the copyright notice ◁
```

This code is used in sections 3 and 59.

61. We must scan through the list of parameters, given in *argv*. The number is in *argc*. We must pay attention to the flag parameter. We need at least 3 parameters (`-m` or `-c`, an output file and a master file) and can handle up to *max_file_index* change files. The names of the file parameters will be inserted into the structure of *input_organisation*. The first file is special. It indicates the output file. When we allow flags at any position, we must find out which name is for what purpose. The master file is already part of the *input_organisation* structure (index 0). As long as the number of files found (counted in *no_ch*) is -1 we have not yet found the output file name.

```
⟨ Scan the parameters 61 ⟩ ≡
  {
    if (argc > max_file_index + 5 − 1) usage_error( );
    no_ch ← −1;      ▷ fill this part of input_organisation ◁
    while (−−argc > 0) {
      argv ++;
      if (strcmp("-help", ∗argv) ≡ 0 ∨ strcmp("--help", ∗argv) ≡ 0) ⟨ Display help message and exit 64 ⟩;
      if (strcmp("-version", ∗argv) ≡ 0 ∨ strcmp("--version", ∗argv) ≡ 0)
        ⟨ Display version information and exit 65 ⟩;
      if (∗∗argv ≡ '-') ⟨ Set a flag 62 ⟩
      else ⟨ Get a file name 63 ⟩
    }
    if (no_ch ≤ 0 ∨ prod_chf ≡ unknown) usage_error( );
  }
```

This code is used in section 3.

62. The flag is about to determine the processing mode. We must make sure that this flag has not been set before. Further flags might be introduced to avoid/force overwriting of output files. Currently we just have to set the processing flag properly.

```
⟨ Set a flag 62 ⟩ ≡
  if (prod_chf ≠ unknown) usage_error( );
  else
    switch (∗(∗argv + 1)) {
    case 'c': case 'C': prod_chf ← chf; break;
    case 'm': case 'M': prod_chf ← master; break;
    default: usage_error( );
    }
```

This code is used in section 61.

63. We have to distinguish whether this is the very first file name (which is the case if *no_ch* ≡ (−1)) or if the next element of *input_organisation* must be filled.

```
⟨ Get a file name 63 ⟩ ≡
  {
    if (no_ch ≡ (−1)) {
      out_name ← ∗argv;
    }
    else {
      register input_description ∗inp_desc;
      inp_desc ← (input_description ∗) malloc(sizeof(input_description));
      if (inp_desc ≡ Λ) fatal_error(−1, "!␣No␣memory␣for␣input␣descriptor", "");
      inp_desc→mode ← search; inp_desc→line ← 0; inp_desc→type_of_file ← chf;
      inp_desc→limit ← inp_desc→buffer; inp_desc→buffer[0] ← '␣'; inp_desc→loc ← inp_desc→buffer + 1;
      inp_desc→buffer_end ← inp_desc→buffer + buf_size − 2; inp_desc→file_name ← ∗argv;
      inp_desc→current_include ← Λ; input_organisation[no_ch] ← inp_desc;
    }
    no_ch++;
  }
```

This code is used in section 61.

64. Modules for dealing with help messages and version info. We follow the `kpathsea` standard code here, so that we can easily adapt this to work with `kpathsea`.

```
⟨ Display help message and exit 64 ⟩ ≡
  usage_help( );
```

This code is used in section 61.

65.

```
⟨ Display version information and exit 65 ⟩ ≡
  {
    print_version_and_exit("CTIE", version_number);
  }
```

This code is used in section 61.

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

```
⟨ Global variables 7 ⟩ +≡
  string CTIEHELP[ ] ← {"Usage:␣ctie␣-[mc]␣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 5 ⟩ +≡
  void usage_help( );
  void print_version_and_exit( );
```

68.
```
void usage_help( )
{
  string *message ← CTIEHELP;
  while (*message) {
    fputs(*message, stdout); putchar('\n'); ++message;
  }
  putchar('\n'); exit(0);
}
```

69.
```
void print_version_and_exit(name, version)
  string name, version;
{
  printf("%s␣%s\n", name, version); puts("Copyright␣(C)␣2002,2003␣Julian␣Gilbey.");
  puts("There␣is␣NO␣warranty.␣␣This␣is␣free␣software.␣␣See␣the␣source");
  puts("code␣of␣CTIE␣for␣redistribution␣conditions."); exit(0);
}
```

70. System-dependent changes. This section should be replaced, if necessary, by changes to the program that are necessary to make `CTIE` work at a particular installation. It is usually best to design your change file so that all changes to previous modules preserve the module numbering; then everybody's version will be consistent with the printed program. More extensive changes, which introduce new modules, can be inserted here; then only the index itself will get a new module number.

71. Index.

⟨ Check for `@i` in newly read line, **goto** *restart* if include fails 26 ⟩ Used in section 19.
⟨ Check that all changes have been read 58 ⟩ Used in section 3.
⟨ Check the current files for any ends of changes 50 ⟩ Used in section 49.
⟨ Display help message and exit 64 ⟩ Used in section 61.
⟨ Display version information and exit 65 ⟩ Used in section 61.
⟨ Error handling functions 29, 36 ⟩ Used in section 2.
⟨ Get a file name 63 ⟩ Used in section 61.
⟨ Get include line into buffer or **goto** *restart* if end of file 24 ⟩ Used in section 19.
⟨ Get line into buffer, **return** *false* if end of file 20 ⟩ Used in section 19.
⟨ Get the master file started 40 ⟩ Used in section 3.
⟨ Global **#include**s 8, 9, 37 ⟩ Used in section 2.
⟨ Global types 4, 10, 11, 12, 13, 14 ⟩ Used in section 2.
⟨ Global variables 7, 15, 16, 18, 22, 39, 66 ⟩ Used in section 2.
⟨ Handle end of file and return 21 ⟩ Used in section 20.
⟨ Handle end of include file and **goto** *restart* 25 ⟩ Used in section 24.
⟨ Handle output 52 ⟩ Used in section 49.
⟨ Increment the line number and print a progess report at certain times 23 ⟩ Used in section 20.
⟨ Initialise parameters 17 ⟩ Used in section 3.
⟨ Internal functions 19, 32, 42, 43, 46, 47, 48, 59 ⟩ Used in section 2.
⟨ Predeclaration of functions 5, 28, 33, 35, 67 ⟩ Used in section 2.
⟨ Prepare the change files 41 ⟩ Used in section 3.
⟨ Prepare the output file 38 ⟩ Used in section 3.
⟨ Print error location based on input buffer 30 ⟩ Used in section 29.
⟨ Print the banners 60 ⟩ Used in sections 3 and 59.
⟨ Print the job *history* 34 ⟩ Used in section 32.
⟨ Process a line, **break** when end of source reached 49 ⟩ Used in section 57.
⟨ Process the input 57 ⟩ Used in section 3.
⟨ Scan all other files for changes to be done 51 ⟩ Used in section 49.
⟨ Scan the parameters 61 ⟩ Used in section 3.
⟨ Set a flag 62 ⟩ Used in section 61.
⟨ Skip over comment lines; **return** if end of file 44 ⟩ Used in section 43.
⟨ Skip to the next nonblank line; **return** if end of file 45 ⟩ Used in section 43.
⟨ Step to next line 56 ⟩ Used in section 49.
⟨ Test for *normal*, **break** when done 53 ⟩ Used in section 52.
⟨ Test for *post*, **break** when done 55 ⟩ Used in section 52.
⟨ Test for *pre*, **break** when done 54 ⟩ Used in section 52.
⟨ The main function 3 ⟩ Used in section 2.
⟨ Try to open include file, abort push if unsuccessful, go to *restart* 27 ⟩ Used in section 26.