The TIE processor

(CWEB Version 2.4)

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. Therefore, we introduce the present program to merge a WEB file and several change files producing a new WEB file. Since the input files are tied together, the program is called TIE. Furthermore, the program can 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. The use of TIE can be expanded to other programming languages since this processor only knows about the structure of change files and does not interpret the master file at all.

The program TIE has to read lines from several input files to bring them in some special ordering. For this purpose an algorithm is used which looks a little bit complicated. But the method used only needs one buffer line for each input file. Thus the storage requirement of TIE does not depend on the input data.

The program is written in C and uses only few features of a particular environment that may need to be changed in other installations. E.g. it will not use the **enum** type declarations. The changes needed may refer to the access of the command line if this can be not supported by any C compiler.

The "banner line" defined here should be changed whenever TIE is modified. This program is put into the public domain. Nevertheless the copyright notice must not be replaced or modified.

```
#define banner "This␣is␣TIE,␣CWEB␣Version␣2.4."
#define copyright "Copyright␣(c)␣1989,1992␣by␣THD/ITI.␣All␣rights␣reserved."
```

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

```
⟨ Global # includes 15 ⟩
⟨ Global constants 5 ⟩
⟨ Global types 4 ⟩
⟨ Global variables 6 ⟩
⟨ Error handling functions 31 ⟩
⟨ Internal functions 24 ⟩
⟨ The main function 59 ⟩
```

3. Here are some macros for common programming idioms.

```
#define incr(v) v += 1 ▷ increase a variable by unity ◁
#define decr(v) v -= 1 ▷ decrease a variable by unity ◁
#define loop while (1) ▷ repeat over and over until a break happens ◁
#define do_nothing ▷ empty statement ◁
  format loop while
```

4. Furthermore we include the additional types *boolean* and *string*.

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

See also sections 7, 8, 18, 19, 20, and 21.

This code is used in section 2.

5. The following parameters should be sufficient for most applications of TIE.

⟨ Global constants 5 ⟩ ≡
#**define** *buf_size* 512 ▷ maximum length of one input line ◁
#**define** *max_file_index* 9
▷ we don't think that anyone needs more than 9 change files, but ... just change it ◁

This code is used in section 2.

6. 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 6 ⟩ ≡
static int *history* ← *spotless*;

See also sections 9, 22, 23, 26, and 35.

This code is used in section 2.

7. The character set.
One of the main goals in the design of TIE has been to make it readily portable between a wide variety of computers. Yet TIE by its very nature must use a greater variety of characters than most computer programs deal with, and character encoding is one of the areas in which existing machines differ most widely from each other.

To resolve this problem, all input to TIE is converted to an internal seven-bit code that is essentially standard ASCII, the "American Standard Code for Information Interchange." The conversion is done immediately when each character is read in. Conversely, characters are converted from ASCII to the user's external representation just before they are output. But the algorithm is prepared for the usage of eight-bit data.
Here is a table of the standard visible ASCII codes:

	0	*1*	*2*	*3*	*4*	*5*	*6*	*7*
Ø040	␣	!	"	#	$	%	&	’
Ø050	(	)	*	+	,	-	.	/
Ø060	0	1	2	3	4	5	6	7
Ø070	8	9	:	;	<	=	>	?
Ø100	@	A	B	C	D	E	F	G
Ø110	H	I	J	K	L	M	N	O
Ø120	P	Q	R	S	T	U	V	W
Ø130	X	Y	Z	[	\	]	^	_
Ø140	‘	a	b	c	d	e	f	g
Ø150	h	i	j	k	l	m	n	o
Ø160	p	q	r	s	t	u	v	w
Ø170	x	y	z	{	|	}	~	

(Actually, of course, code °*40* is an invisible blank space.) Code °*136* was once an upward arrow (↑), and code °*137* was once a left arrow (←), in olden times when the first draft of ASCII code was prepared; but TIE works with today's standard ASCII in which those codes represent circumflex and underline as shown. The maximum value used is also defined, it must be changed if an extended ASCII is used.

If the C compiler is not able to process **unsigned char**'s, you should define *ASCII_Code* as **short**.

```
⟨ Global types 4 ⟩ +≡
#define max_ASCII (@'~' + 1)
  typedef unsigned char ASCII_Code;      ▷ eight-bit numbers, a subrange of the integers ◁
```

8. C was first implemented on a machine that uses the ASCII representation for characters. But to make it readily available also for other machines (big brother is watching?) we add a conversion that may be undone for most installations. TIE assumes that it is being used with a character set that contains at least the characters of standard ASCII as listed above.

In order to accommodate this difference, we shall use the name *text_char* to stand for the data type of the characters in the input and output files. We shall also assume that *text_char* consists of the elements *first_text_char* through *last_text_char*, inclusive. The following definitions should be adjusted if necessary.

```
#define first_text_char 0      ▷ ordinal number of the smallest element of text_char ◁
#define last_text_char 255     ▷ ordinal number of the largest element of text_char ◁
⟨ Global types 4 ⟩ +≡
  typedef unsigned char text_char;      ▷ the data type of characters in text files ◁
  typedef FILE *text_file;
```

9. The TIE processor converts between ASCII code and the user's external character set by means of arrays *xord* and *xchr* that are analogous to Pascal's *ord* and *chr* functions.

The mapping may be disabled by changing the following macro definitions to just a cast. If your C compiler does not support **unsigned char**'s, you should incorporate a binary and with `#ff`.

```
#define map_xchr(c) (text_char)(c)   ▷ change this to xchr[c] on non ASCII machines ◁
#define map_xord(c) (ASCII_Code)(c)   ▷ change this to xord[c] on non ASCII machines ◁
⟨ Global variables 6 ⟩ +≡
  static ASCII_Code xord[last_text_char + 1];   ▷ specifies conversion of input characters ◁
  static text_char xchr[max_ASCII + 1];   ▷ specifies conversion of output characters ◁
```

10. If we assume that every system using WEB is able to read and write the visible characters of standard ASCII (although not necessarily using the ASCII codes to represent them), the following assignment statements initialize most of the *xchr* array properly, without needing any system-dependent changes. For example, the statement *xchr*[@'A'] ← 'A' that appears in the present WEB file might be encoded in, say, EBCDIC code on the external medium on which it resides, but CTANGLE will convert from this external code to ASCII and back again. Therefore the assignment statement *xchr*[65] ← 'A' will appear in the corresponding C file, and C will compile this statement so that *xchr*[65] receives the character A in the external code. Note that it would be quite incorrect to say *xchr*[@'A'] ← @'A', because @'A' is a constant of type **int** not **char**, and because we have @'A' ≡ 65 regardless of the external character set.

```
⟨ Set initial values 10 ⟩ ≡
  xchr[@'␣'] ← '␣'; xchr[@'!'] ← '!'; xchr[@'\"'] ← '\"'; xchr[@'#'] ← '#';
  xchr[@'$'] ← '$'; xchr[@'%'] ← '%'; xchr[@'&'] ← '&'; xchr[@'\''] ← '\'';
  xchr[@'('] ← '('; xchr[@')'] ← ')'; xchr[@'*'] ← '*'; xchr[@'+'] ← '+';
  xchr[@','] ← ','; xchr[@'-'] ← '-'; xchr[@'.'] ← '.'; xchr[@'/'] ← '/';
  xchr[@'0'] ← '0'; xchr[@'1'] ← '1'; xchr[@'2'] ← '2'; xchr[@'3'] ← '3';
  xchr[@'4'] ← '4'; xchr[@'5'] ← '5'; xchr[@'6'] ← '6'; xchr[@'7'] ← '7';
  xchr[@'8'] ← '8'; xchr[@'9'] ← '9'; xchr[@':'] ← ':'; xchr[@';'] ← ';';
  xchr[@'<'] ← '<'; xchr[@'='] ← '='; xchr[@'>'] ← '>'; xchr[@'?'] ← '?';
  xchr[@'@'] ← '@'; xchr[@'A'] ← 'A'; xchr[@'B'] ← 'B'; xchr[@'C'] ← 'C';
  xchr[@'D'] ← 'D'; xchr[@'E'] ← 'E'; xchr[@'F'] ← 'F'; xchr[@'G'] ← 'G';
  xchr[@'H'] ← 'H'; xchr[@'I'] ← 'I'; xchr[@'J'] ← 'J'; xchr[@'K'] ← 'K';
  xchr[@'L'] ← 'L'; xchr[@'M'] ← 'M'; xchr[@'N'] ← 'N'; xchr[@'O'] ← 'O';
  xchr[@'P'] ← 'P'; xchr[@'Q'] ← 'Q'; xchr[@'R'] ← 'R'; xchr[@'S'] ← 'S';
  xchr[@'T'] ← 'T'; xchr[@'U'] ← 'U'; xchr[@'V'] ← 'V'; xchr[@'W'] ← 'W';
  xchr[@'X'] ← 'X'; xchr[@'Y'] ← 'Y'; xchr[@'Z'] ← 'Z'; xchr[@'['] ← '[';
  xchr[@'\\'] ← '\\'; xchr[@']'] ← ']'; xchr[@'^'] ← '^'; xchr[@'_'] ← '_';
  xchr[@'‘'] ← '‘'; xchr[@'a'] ← 'a'; xchr[@'b'] ← 'b'; xchr[@'c'] ← 'c';
  xchr[@'d'] ← 'd'; xchr[@'e'] ← 'e'; xchr[@'f'] ← 'f'; xchr[@'g'] ← 'g';
  xchr[@'h'] ← 'h'; xchr[@'i'] ← 'i'; xchr[@'j'] ← 'j'; xchr[@'k'] ← 'k';
  xchr[@'l'] ← 'l'; xchr[@'m'] ← 'm'; xchr[@'n'] ← 'n'; xchr[@'o'] ← 'o';
  xchr[@'p'] ← 'p'; xchr[@'q'] ← 'q'; xchr[@'r'] ← 'r'; xchr[@'s'] ← 's';
  xchr[@'t'] ← 't'; xchr[@'u'] ← 'u'; xchr[@'v'] ← 'v'; xchr[@'w'] ← 'w';
  xchr[@'x'] ← 'x'; xchr[@'y'] ← 'y'; xchr[@'z'] ← 'z'; xchr[@'{'] ← '{';
  xchr[@'|'] ← '|'; xchr[@'}'] ← '}'; xchr[@'~'] ← '~';
  xchr[0] ← '␣'; xchr[#7F] ← '␣';   ▷ these ASCII codes are not used ◁
```

See also sections 13 and 14.

This code is used in section 59.

11. Some of the ASCII codes below `#20` have been given a symbolic name in `TIE` because they are used with a special meaning.

```
#define tab_mark  @'\t'    ▷ ASCII code used as tab-skip ◁
#define nl_mark  @'\n'    ▷ ASCII code used as line end marker ◁
#define form_feed  @'\f'    ▷ ASCII code used as page eject ◁
```

12. When we initialize the *xord* array and the remaining parts of *xchr*, it will be convenient to make use of an index variable, *i*.

```
⟨ Local variables for initialisation 12 ⟩ ≡
  int i;
```

This code is used in section 59.

13. Here now is the system-dependent part of the character set. If `TIE` is being implemented on a garden-variety C for which only standard ASCII codes will appear in the input and output files, you don't need to make any changes here.

Changes to the present module will make `TIE` more friendly on computers that have an extended character set, so that one can type things like ≠. If you have an extended set of characters that are easily incorporated into text files, you can assign codes arbitrarily here, giving an *xchr* equivalent to whatever characters the users of `TIE` are allowed to have in their input files, provided that unsuitable characters do not correspond to special codes like *tab_mark* that are listed above.

```
⟨ Set initial values 10 ⟩ +≡
  for (i ← 1; i < @'␣'; xchr[i++] ← '␣') ;
  xchr[tab_mark] ← '\t'; xchr[form_feed] ← '\f'; xchr[nl_mark] ← '\n';
```

14. The following system-independent code makes the *xord* array contain a suitable inverse to the information in *xchr*.

```
⟨ Set initial values 10 ⟩ +≡
  for (i ← first_text_char; i ≤ last_text_char; xord[i++] ← @'␣') do_nothing;
  for (i ← 1; i ≤ @'~'; i++) xord[xchr[i]] ← i;
```

15. Input and output.
Output for the user is done by writing on file *term_out*, which is assumed to consist of characters of type **text_char**. It should be linked to *stdout* usually. Terminal input is not needed in this version of `TIE`. *stdin* and *stdout* are predefined if we include the `stdio.h` definitions. Although I/O redirection for *stdout* is usually available you may lead output to another file if you change the definition of *term_out*. Also we define some macros for terminating an output line and writing strings to the user.

```
#define term_out stdout
#define print(a) fprintf(term_out,a)  ▷ 'print' means write on the terminal ◁
#define print2(a,b) fprintf(term_out,a,b)  ▷ same with two arguments ◁
#define print3(a,b,c) fprintf(term_out,a,b,c)  ▷ same with three arguments ◁
#define print_c(v) fputc(v,term_out);  ▷ print a single character ◁
#define new_line(v) fputc('\n',v)  ▷ start new line ◁
#define term_new_line new_line(term_out)  ▷ start new line of the terminal ◁
#define print_ln(v)
        {
          fprintf(term_out,v); term_new_line;
        }  ▷ 'print' and then start new line ◁
#define print2_ln(a,b)
        {
          print2(a,b); term_new_line;
        }  ▷ same with two arguments ◁
#define print3_ln(a,b,c)
        {
          print3(a,b,c); term_new_line;
        }  ▷ same with three arguments ◁
#define print_nl(v)
        {
          term_new_line; print(v);
        }  ▷ print information starting on a new line ◁
#define print2_nl(a,b)
        {
          term_new_line; print2(a,b);
        }  ▷ same for two arguments ◁
```

⟨ Global # **include**s 15 ⟩ ≡
```
#include <stdio.h>
```
See also section 16.
This code is used in section 2.

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

⟨ Global # **include**s 15 ⟩ +≡
```
#ifdef __STDC__
#include <stdlib.h>
#else
#include <malloc.h>
#endif
```

17. The *update_terminal* function is called when we want to make sure that everything we have output to the terminal so far has actually left the computer's internal buffers and been sent.

```
#define update_terminal fflush(term_out)  ▷ empty the terminal output buffer ◁
```

18. Data structures.
The multiple 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 `TIE` 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) ◁
```

19. 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) ◁
```

20. Two more types will indicate variables used as an index into either the file buffer or the file table.

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

21. The following data structure joins all informations needed to use these input files.

```
  format line dummy
⟨ Global types 4 ⟩ +≡
  typedef struct _idsc {
    string name_of_file;
    ASCII_Code buffer[buf_size];
    in_file_modes mode;
    long line;
    file_types type_of_file;
    buffer_index limit;
    text_file the_file;
  } input_description;
```

22. These data types are used in the global variable section. They refer to the files in action, the number of change files, the mode of operation and the current output state.

```
⟨ Global variables 6 ⟩ +≡
  static file_index actual_input, test_input, no_ch;
  static file_types prod_chf ← unknown;
  static out_md_type out_mode;
```

23. All input files (including the master file) are recorded in the following structure. Mostly the components are accessed through a local pointer variable. This corresponds to Pascal's **with**-statement and results in a one-time-computation of the index expression.

```
⟨ Global variables 6 ⟩ +≡
  static input_description *input_organization[max_file_index + 1];
```

24. 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.

```
⟨ Internal functions 24 ⟩ ≡
  void get_line(i)
      file_index i;
  {
    register input_description *inp_desc ← input_organization[i];
    if (inp_desc→mode ≡ ignore) return;
    if (feof(inp_desc→the_file)) ⟨ Handle end of file and return 25 ⟩
    ⟨ Get line into buffer 27 ⟩
  }
```

See also sections 38, 39, 42, 43, 44, and 55.

This code is used in section 2.

25. 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 25 ⟩ ≡
  {
    inp_desc→mode ← ignore; inp_desc→limit ← −1;      ▷ mark end-of-file ◁
    if (inp_desc→type_of_file ≡ master) input_has_ended ← true;
    return;
  }
```

This code is used in section 24.

26. This variable must be declared for global access.

```
⟨ Global variables 6 ⟩ +≡
  static boolean input_has_ended ← false;
```

27. 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 27 ⟩ ≡
  {
    int final_limit;      ▷ used to delete trailing spaces ◁
    int c;      ▷ the actual character read ◁
    ⟨ Increment the line number and print a progess report at certain times 28 ⟩
    inp_desc→limit ← final_limit ← 0;
    while (inp_desc→limit < buf_size) {
      c ← fgetc(inp_desc→the_file);
      ⟨ Check c for EOF, return if line was empty, otherwise break to process last line 29 ⟩
      inp_desc→buffer[inp_desc→limit++] ← c ← map_xord(c);
      if (c ≡ nl_mark) break;      ▷ end of line found ◁
      if (c ≠ @'␣' ∧ c ≠ tab_mark) final_limit ← inp_desc→limit;
    }
    ⟨ Test for truncated line, skip to end of line 30 ⟩
    inp_desc→limit ← final_limit;
  }
```

This code is used in section 24.

28. 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 28 ⟩ ≡
  incr(inp_desc→line);
  if (inp_desc→type_of_file ≡ master ∧ inp_desc→line % 100 ≡ 0) {
    if (inp_desc→line % 500 ≡ 0) print2("%ld", inp_desc→line);
    else print_c('.');
    update_terminal;
  }
```

This code is used in section 27.

29. There may be incomplete lines of the editor used does not make sure that the last character before end of file is an end of line. In such a case we must process the final line. Of the current line is empty, we just can **return**. Note that this test must be done *before* the character read is translated.

```
⟨ Check c for EOF, return if line was empty, otherwise break to process last line 29 ⟩ ≡
  if (c ≡ EOF) {
    if (inp_desc→limit ≤ 0) {
      inp_desc→mode ← ignore; inp_desc→limit ← −1;      ▷ mark end-of-file ◁
      if (inp_desc→type_of_file ≡ master) input_has_ended ← true;
      return;
    }
    else {      ▷ add end of line mark ◁
      c ← nl_mark; break;
    }
  }
```

This code is used in section 27.

30. If the line is truncated we must skip the rest (and still watch for `EOF`!).

```
⟨ Test for truncated line, skip to end of line 30 ⟩ ≡
  if (c ≠ nl_mark) {
    err_print("! Input line too long")(i);
    while ((c ← fgetc(inp_desc¬the_file)) ≠ EOF ∧ map_xord(c) ≠ nl_mark) do_nothing;      ▷ skip to end ◁
  }
```

This code is used in section 27.

31. 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

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

where *file_no* is the number of the file which is concerned by the error. Please note that no trailing dot is supplied by the error message because it is appended by *err_print*.

This function is implemented as a macro. It gives a message and an indication of the offending file. The actions to determine the error location are provided by a function called *err_loc*.

```
#define error_loc(m)  err_loc(m); history ← troublesome; }
#define err_print(m)  { print_nl(m); error_loc
```

⟨ Error handling functions 31 ⟩ ≡

```
void err_loc(i)      ▷ prints location of error ◁
    int i;
{
  print3_ln("␣(file␣%s,␣l.%ld).", input_organization[i]→name_of_file, input_organization[i]→line);
}
```

This code is used in section 2.

32. Non recoverable errors are handled by calling *fatal_error* that outputs a message and then calls '*jump_out*'. *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(m)
        {
          print(m); print_c('.'); history ← fatal; term_new_line; jump_out( );
        }
```

33. *jump_out* just cuts across all active procedure levels and jumps out of the program. It is used when no recovery from a particular error has been provided. The return code from this program should be regarded by the caller.

```
#define jump_out()  exit(1)
```

34. 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 34 ⟩ ≡
  {
    out_file ← fopen(out_name, "w");
    if (out_file ≡ Λ) {
      fatal_error("! Could not open/create output file");
    }
  }
```

This code is used in section 59.

35. The name of the file and the file desciptor are stored in global variables.

```
⟨ Global variables 6 ⟩ +≡
  static text_file out_file;
  static string out_name;
```

36. For the master file we start just reading its first line into the buffer, if we could open it.

```
⟨ Get the master file started 36 ⟩ ≡
  {
    input_organization[0]→the_file ← fopen(input_organization[0]→name_of_file, "r");
    if (input_organization[0]→the_file ≡ Λ) fatal_error("! Could not open master file");
    print2("(%s)", input_organization[0]→name_of_file); term_new_line;
    input_organization[0]→type_of_file ← master; get_line(0);
  }
```

This code is used in section 59.

37. For the change files we must skip the comment part and see, whether we can find any change in it. This is done by *init_change_file*.

```
⟨ Prepare the change files 37 ⟩ ≡
  {
    file_index i;
    i ← 1;
    while (i < no_ch) {
      input_organization[i]→the_file ← fopen(input_organization[i]→name_of_file, "r");
      if (input_organization[i]→the_file ≡ Λ) fatal_error("!Could not open change file");
      print2("(%s)", input_organization[i]→name_of_file); term_new_line; init_change_file(i, true); incr(i);
    }
  }
```

This code is used in section 59.

38. Input/output organization.
Here's a simple function that checks if two lines are different.

```
⟨ Internal functions 24 ⟩ +≡
  boolean lines_dont_match(i, j)
      file_index i, j;
  {
    buffer_index k, lmt;
    if (input_organization[i]¬limit ≠ input_organization[j]¬limit) return (true);
    lmt ← input_organization[i]¬limit;
    for (k ← 0; k < lmt; k++)
      if (input_organization[i]¬buffer[k] ≠ input_organization[j]¬buffer[k]) return (true);
    return (false);
  }
```

39. Function *init_change_file*(i, b) is used to ignore all lines of the input file with index i until the next change module is found. The boolean parameter b indicates whether we do not want to see `@x` or `@y` entries during our skip.

```
⟨ Internal functions 24 ⟩ +≡
  void init_change_file(i, b)
      file_index i;
      boolean b;
  {
    register input_description *inp_desc ← input_organization[i];
    ⟨ Skip over comment lines; return if end of file 40 ⟩
    ⟨ Skip to the next nonblank line; return if end of file 41 ⟩
  }
```

40. 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` or `@z` (which would probably indicate that the change file is fouled up).

```
⟨ Skip over comment lines; return if end of file 40 ⟩ ≡
  loop {
    ASCII_Code c;
    get_line(i);
    if (inp_desc¬mode ≡ ignore) return;
    if (inp_desc¬limit < 2) continue;
    if (inp_desc¬buffer[0] ≠ '@') continue;
    c ← inp_desc¬buffer[1];
    if (c ≥ 'X' ∧ c ≤ 'Z') c += 'z' − 'Z';      ▷ lowercasify ◁
    if (c ≡ 'x') break;
    if (c ≡ 'y' ∨ c ≡ 'z')
      if (b)      ▷ scanning for start of change ◁
        err_print("! Where is the matching @x?")(i);
  }
```

This code is used in section 39.

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

```
⟨ Skip to the next nonblank line; return if end of file 41 ⟩ ≡
  do {
    get_line(i);
    if (inp_desc→mode ≡ ignore) {
      err_print("!␣Change␣file␣ended␣after␣@x")(i); return;
    }
  } while (inp_desc→limit ≤ 0);
```

This code is used in section 39.

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

```
⟨ Internal functions 24 ⟩ +≡
  void put_line(j)
      file_index j;
  {
    buffer_index i;      ▷ index into the buffer ◁
    buffer_index lmt;    ▷ line length ◁
    ASCII_Code *p;       ▷ output pointer ◁
    lmt ← input_organization[j]→limit; p ← input_organization[j]→buffer;
    for (i ← 0; i < lmt; i++) fputc(map_xchr(*p++), out_file);
    new_line(out_file);
  }
```

43. The function *e_of_ch_module* returns true if the input line from file i starts with `@z`.

```
⟨ Internal functions 24 ⟩ +≡
  boolean e_of_ch_module(i)
      file_index i;
  {
    register input_description *inp_desc ← input_organization[i];
    if (inp_desc→limit < 0) {
      print_nl("!␣At␣the␣end␣of␣change␣file␣missing␣@z␣");
      print2("%s", input_organization[i]→name_of_file); term_new_line; return (true);
    }
    else if (inp_desc→limit ≥ 2)
      if (inp_desc→buffer[0] ≡ '@' ∧ (inp_desc→buffer[1] ≡ 'Z' ∨ inp_desc→buffer[1] ≡ 'z'))
        return (true);
    return (false);
  }
```

44. The function *e_of_ch_preamble* returns *true* if the input line from file i starts with `@y`.

```
⟨ Internal functions 24 ⟩ +≡
  boolean e_of_ch_preamble(i)
      file_index i;
  {
    register input_description *inp_desc ← input_organization[i];
    if (inp_desc→limit ≥ 2 ∧ inp_desc→buffer[0] ≡ '@')
      if (inp_desc→buffer[1] ≡ 'Y' ∨ inp_desc→buffer[1] ≡ 'y') return (true);
    return (false);
  }
```

45. To process the input file the next section reads a line of the actual input file and updates the *input_organization* for all files with index *test_file* greater *actual_input*.

```
⟨ Process a line, break when end of source reached 45 ⟩ ≡
  {
    file_index test_file;
    ⟨ Check the current files for any ends of changes 46 ⟩
    if (input_has_ended ∧ actual_input ≡ 0) break;      ▷ all done ◁
    ⟨ Scan all other files for changes to be done 47 ⟩
    ⟨ Handle output 48 ⟩
    ⟨ Step to next line 52 ⟩
  }
```

This code is used in section 53.

46. Any of the current change files may have reached the end of 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 inactive if it reaches end of file.

```
⟨ Check the current files for any ends of changes 46 ⟩ ≡
  {
    register input_description *inp_desc;
    while (actual_input > 0 ∧ e_of_ch_module(actual_input)) {
      inp_desc ← input_organization[actual_input];
      if (inp_desc→type_of_file ≡ master) {      ▷ emergency exit, everything mixed up! ◁
        fatal_error("!␣This␣can't␣happen:␣change␣file␣is␣master␣file");
      }
      inp_desc→mode ← search; init_change_file(actual_input, true);
      while ((input_organization[actual_input]→mode ≠ reading ∧ actual_input > 0)) decr(actual_input);
    }
  }
```

This code is used in section 45.

47. Now we will set *test_input* to the file that has another match for the current line. This depends on the state of the other change files. If no other file matches, *actual_input* refers to a line to write and *test_input* ist set to *none*.

```
#define none  (max_file_index + 1)
⟨ Scan all other files for changes to be done 47 ⟩ ≡
  test_input ← none; test_file ← actual_input;
  while (test_input ≡ none ∧ test_file < no_ch − 1) {
    incr(test_file);
    switch (input_organization[test_file]→mode) {
    case search:
      if (lines_dont_match(actual_input, test_file) ≡ false) {
        input_organization[test_file]→mode ← test; test_input ← test_file;
      }
      break;
    case test:
      if (lines_dont_match(actual_input, test_file) ≡ true) {      ▷ error, sections do not match ◁
        input_organization[test_file]→mode ← search;
        err_print("! Sections do not match")(actual_input); err_loc(test_file);
        init_change_file(test_file, false);
      }
      else test_input ← test_file;
      break;
    case reading: do_nothing;      ▷ this can't happen ◁
      break;
    case ignore: do_nothing;      ▷ nothing to do ◁
      break;
    }
  }
```

This code is used in section 45.

48. For the output we must distinguish whether we create a new change file or a new master file. The change file creation needs some closer inspection because we may be before a change, in the pattern part or in the replacement part. For a master file we have to write the line from the current actual input.

```
⟨ Handle output 48 ⟩ ≡
  if (prod_chf ≡ chf) { loop {
    ⟨ Test for normal, break when done 49 ⟩
    ⟨ Test for pre, break when done 50 ⟩
    ⟨ Test for post, break when done 51 ⟩
  }
  }
  else
    if (test_input ≡ none) put_line(actual_input);
```

This code is used in section 45.

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

```
⟨ Test for normal, break when done 49 ⟩ ≡
  if (out_mode ≡ normal) {
    if (test_input ≠ none) {
      fputc(map_xchr('@'@'), out_file); fputc(map_xchr('@'x'), out_file); new_line(out_file);
      out_mode ← pre;
    }
    else break;
  }
```

This code is used in section 48.

50. Check whether we have to start the replacement text. This is the case when we have no more matching line. Otherwise the master file source line must be copied to the change file.

```
⟨ Test for pre, break when done 50 ⟩ ≡
  if (out_mode ≡ pre) {
    if (test_input ≡ none) {
      fputc(map_xchr('@'@'), out_file); fputc(map_xchr('@'y'), out_file); new_line(out_file);
      out_mode ← post;
    }
    else {
      if (input_organization[actual_input]→type_of_file ≡ master) put_line(actual_input);
      break;
    }
  }
```

This code is used in section 48.

51. Check whether an entry from a change file is complete. If the current change was a change for a change file in effect, 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 51 ⟩ ≡
  if (out_mode ≡ post) {
    if (input_organization[actual_input]→type_of_file ≡ chf) {
      if (test_input ≡ none) put_line(actual_input);
      break;
    }
    else {
      fputc(map_xchr('@'@'), out_file); fputc(map_xchr('@'z'), out_file); new_line(out_file);
      new_line(out_file); out_mode ← normal;
    }
  }
```

This code is used in section 48.

52. 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 52 ⟩ ≡
  get_line(actual_input);
  if (test_input ≠ none) {
    get_line(test_input);
    if (e_of_ch_preamble(test_input) ≡ true) {
      get_line(test_input);      ▷ update current changing file ◁
      input_organization[test_input]→mode ← reading; actual_input ← test_input; test_input ← none;
    }
  }
```

This code is used in section 45.

53. 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 53 ⟩ ≡
  actual_input ← 0; input_has_ended ← false;
  while (input_has_ended ≡ false ∨ actual_input ≠ 0)
    ⟨ Process a line, break when end of source reached 45 ⟩
  if (out_mode ≡ post) {      ▷ last line has been changed ◁
    fputc(map_xchr('@'@'), out_file); fputc(map_xchr('@'z'), out_file); new_line(out_file);
  }
```

This code is used in section 59.

54. 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 54 ⟩ ≡
  {
    file_index i;
    for (i ← 1; i < no_ch; i++) {      ▷ all change files ◁
      if (input_organization[i]→mode ≠ ignore) err_print("! Change file entry did not match")(i);
    }
  }
```

This code is used in section 59.

55. We want to tell the user about our command line options. This is done by the *usage*( ) function. It contains merely the necessary print statement and exits afterwards.

```
⟨ Internal functions 24 ⟩ +≡
  void usage( )
  {
    print("Usage: tie -[mc] outfile master changefile(s)"); term_new_line; jump_out( );
  }
```

56. 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 5 parameters and can handle up to *max_file_index* change files. The names fo the file parameters will be inserted into the structure of *input_organization*. 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_organization* 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 56 ⟩ ≡
  {
    int act_arg;
    if (argc < 5 ∨ argc > max_file_index + 4 − 1) usage( );
    no_ch ← −1;      ▷ fill this part of input_organization ◁
    for (act_arg ← 1; act_arg < argc; act_arg++) {
      if (argv[act_arg][0] ≡ '-') ⟨ Set a flag 57 ⟩
      else ⟨ Get a file name 58 ⟩
    }
    if (no_ch ≤ 0 ∨ prod_chf ≡ unknown) usage( );
  }
```

This code is used in section 59.

57. 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 57 ⟩ ≡
  if (prod_chf ≠ unknown) usage( );
  else
    switch (argv[act_arg][1]) {
    case 'c': case 'C': prod_chf ← chf; break;
    case 'm': case 'M': prod_chf ← master; break;
    default: usage( );
    }
```

This code is used in section 56.

58. We have to distinguish whether this is the very first file name (known if *no_ch* $\equiv (-1)$) or if the next element of *input_organization* must be filled.

```
⟨ Get a file name 58 ⟩ ≡
  {
    if (no_ch ≡ (−1)) {
      out_name ← argv[act_arg];
    }
    else {
      register input_description *inp_desc;
      inp_desc ← (input_description *) malloc(sizeof(input_description));
      if (inp_desc ≡ Λ) fatal_error("! No memory for descriptor");
      inp_desc→mode ← search; inp_desc→line ← 0; inp_desc→type_of_file ← chf; inp_desc→limit ← 0;
      inp_desc→name_of_file ← argv[act_arg]; input_organization[no_ch] ← inp_desc;
    }
    incr(no_ch);
  }
```

This code is used in section 56.

59. The main program.
Here is where `TIE` starts, and where it ends.

```
⟨ The main function 59 ⟩ ≡
  main(argc, argv)
      int argc;
      string *argv;
  {
    {
      ⟨ Local variables for initialisation 12 ⟩
      ⟨ Set initial values 10 ⟩
    }
    print_ln(banner);      ▷ print a "banner line" ◁
    print_ln(copyright);      ▷ include the copyright notice ◁
    actual_input ← 0; out_mode ← normal; ⟨ Scan the parameters 56 ⟩
    ⟨ Prepare the output file 34 ⟩
    ⟨ Get the master file started 36 ⟩
    ⟨ Prepare the change files 37 ⟩
    ⟨ Process the input 53 ⟩
    ⟨ Check that all changes have been read 54 ⟩
    ⟨ Print the job history 60 ⟩
  }
```

This code is used in section 2.

60. We want to pass the *history* value to the operating system so that it can be used to govern whether or not other programs are started. Additionaly we report the history to the user, although this may not be "UNIX" style—but we are in best companion: `WEB` and TEX do the same.

```
⟨ Print the job history 60 ⟩ ≡
  {
    string msg;
    switch (history) {
    case spotless: msg ← "No␣errors␣were␣found"; break;
    case troublesome: msg ← "Pardon␣me,␣but␣I␣think␣I␣spotted␣something␣wrong."; break;
    case fatal: msg ← "That␣was␣a␣fatal␣error,␣my␣friend"; break;
    }      ▷ there are no other cases ◁
    print2_nl("(%s.)", msg); term_new_line; exit(history ≡ spotless ? 0 : 1);
  }
```

This code is used in section 59.

61. System-dependent changes.
This section should be replaced, if necessary, by changes to the program that are necessary to make `TIE` 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.

62. Index.
Here is the cross-reference table for the TIE processor.

⟨ Check that all changes have been read 54 ⟩ Used in section 59.
⟨ Check the current files for any ends of changes 46 ⟩ Used in section 45.
⟨ Check *c* for `EOF`, **return** if line was empty, otherwise **break** to process last line 29 ⟩ Used in section 27.
⟨ Error handling functions 31 ⟩ Used in section 2.
⟨ Get a file name 58 ⟩ Used in section 56.
⟨ Get line into buffer 27 ⟩ Used in section 24.
⟨ Get the master file started 36 ⟩ Used in section 59.
⟨ Global constants 5 ⟩ Used in section 2.
⟨ Global types 4, 7, 8, 18, 19, 20, 21 ⟩ Used in section 2.
⟨ Global variables 6, 9, 22, 23, 26, 35 ⟩ Used in section 2.
⟨ Global # **include**s 15, 16 ⟩ Used in section 2.
⟨ Handle end of file and return 25 ⟩ Used in section 24.
⟨ Handle output 48 ⟩ Used in section 45.
⟨ Increment the line number and print a progess report at certain times 28 ⟩ Used in section 27.
⟨ Internal functions 24, 38, 39, 42, 43, 44, 55 ⟩ Used in section 2.
⟨ Local variables for initialisation 12 ⟩ Used in section 59.
⟨ Prepare the change files 37 ⟩ Used in section 59.
⟨ Prepare the output file 34 ⟩ Used in section 59.
⟨ Print the job *history* 60 ⟩ Used in section 59.
⟨ Process a line, **break** when end of source reached 45 ⟩ Used in section 53.
⟨ Process the input 53 ⟩ Used in section 59.
⟨ Scan all other files for changes to be done 47 ⟩ Used in section 45.
⟨ Scan the parameters 56 ⟩ Used in section 59.
⟨ Set a flag 57 ⟩ Used in section 56.
⟨ Set initial values 10, 13, 14 ⟩ Used in section 59.
⟨ Skip over comment lines; **return** if end of file 40 ⟩ Used in section 39.
⟨ Skip to the next nonblank line; **return** if end of file 41 ⟩ Used in section 39.
⟨ Step to next line 52 ⟩ Used in section 45.
⟨ Test for normal, **break** when done 49 ⟩ Used in section 48.
⟨ Test for post, **break** when done 51 ⟩ Used in section 48.
⟨ Test for pre, **break** when done 50 ⟩ Used in section 48.
⟨ Test for truncated line, skip to end of line 30 ⟩ Used in section 27.
⟨ The main function 59 ⟩ Used in section 2.