The CTANGLE processor

(Version 4.12.2 [TEX Live])

Copyright © 1987, 1990, 1993, 2000 Silvio Levy and Donald E. Knuth

Permission is granted to make and distribute verbatim copies of this document provided that the copyright notice and this permission notice are preserved on all copies.

Permission is granted to copy and distribute modified versions of this document under the conditions for verbatim copying, provided that the entire resulting derived work is given a different name and distributed under the terms of a permission notice identical to this one.

Editor's Note: The present variant of this C/WEB source file has been modified for use in the TEX Live system.

The following sections were changed by the change file: 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 20, 29, 35, 40, 47, 48, 50, 54, 59, 67, 74, 75, 76, 79, 80, 81, 82, 83, 86, 87, 88, 89, 93, 100, 102, 103, 104, 105, 106, 107, 108, 109, 110, 111, 112, 113, 114, 115, 116, 117, 118.

March 2, 2026 at 08:33

1* Introduction. This is the CTANGLE program by Silvio Levy and Donald E. Knuth, based on TANGLE by Knuth. We are thankful to Nelson Beebe, Hans-Hermann Bode (to whom the C++ adaptation is due), Klaus Guntermann, Norman Ramsey, Tomas Rokicki, Joachim Schnitter, Joachim Schrod, Lee Wittenberg, and others who have contributed improvements.

The "banner line" defined here should be changed whenever CTANGLE is modified.

```
#define banner "This␣is␣CTANGLE,␣Version␣4.12.2"
        ▷ will be extended by the TEX Live versionstring ◁
```

⟨ Include files 5* ⟩
⟨ Preprocessor definitions ⟩
⟨ Common code for CWEAVE and CTANGLE 3* ⟩
⟨ Typedef declarations 19 ⟩
⟨ Private variables 20* ⟩
⟨ Predeclaration of procedures 4* ⟩

2* CTANGLE has a fairly straightforward outline. It operates in two phases: First it reads the source file, saving the C code in compressed form; then it shuffles and outputs the code.

Please read the documentation for COMMON, the set of routines common to CTANGLE and CWEAVE, before proceeding further.

```
int main(int ac, char **av)
{
  argc ← ac; argv ← av; program ← ctangle; ⟨ Set initial values 21 ⟩
  common_init();
  if (show_banner) cb_show_banner();    ▷ print a "banner line" ◁
  phase_one();    ▷ read all the user's text and compress it into tok_mem ◁
  phase_two();    ▷ output the contents of the compressed tables ◁
  return wrap_up();    ▷ and exit gracefully ◁
}
```

3* The next few sections contain stuff from the file "common.w" that must be included in both "ctangle.w" and "cweave.w". It appears in file "common.h", which is also included in "common.w" to propagate possible changes from this COMMON interface consistently.

First comes general stuff:

⟨ Common code for CWEAVE and CTANGLE 3* ⟩ ≡

```
typedef uint8_t eight_bits;
typedef uint16_t sixteen_bits;
typedef enum {
  ctangle, cweave, ctwill
} cweb;
extern cweb program;    ▷ CTANGLE or CWEAVE or CTWILL? ◁
extern int phase;    ▷ which phase are we in? ◁
```

See also sections 6*, 7*, 8*, 10*, 11*, 13*, 15*, 16*, and 116*.

This code is used in section 1*.

4* The procedure that gets everything rolling:

⟨ Predeclaration of procedures 4* ⟩ ≡

```
extern void common_init(void);
```

See also sections 9*, 12*, 14*, 30, 37, 44, 49, 65, 70, 84, 91, 99, and 101.

This code is used in section 1*.

5* You may have noticed that almost all `"strings"` in the CWEB sources are placed in the context of the '_' macro. This is just a shortcut for the '*gettext*' function from the "GNU gettext utilities." For systems that do not have this library installed, we wrap things for neutral behavior without internationalization.

```
#define _(s) gettext(s)
⟨ Include files 5* ⟩ ≡
#include <ctype.h>    ▷ definition of isalpha, isdigit and so on ◁
#include <stdbool.h>    ▷ definition of bool, true and false ◁
#include <stddef.h>    ▷ definition of ptrdiff_t ◁
#include <stdint.h>    ▷ definition of uint8_t and uint16_t ◁
#include <stdio.h>    ▷ definition of printf and friends ◁
#include <stdlib.h>    ▷ definition of getenv and exit ◁
#include <string.h>    ▷ definition of strlen, strcmp and so on ◁
#ifndef HAVE_GETTEXT
#define HAVE_GETTEXT 0
#endif
#if HAVE_GETTEXT
#include <libintl.h>
#else
#define gettext(a) a
#endif
```

This code is used in section 1*.

6* Code related to the character set:

```
#define and_and °4    ▷ '&&'; corresponds to MIT's ∧ and ASCII EOT ◁
#define lt_lt °20    ▷ '<<'; corresponds to MIT's ⊂ and ASCII DLE ◁
#define gt_gt °21    ▷ '>>'; corresponds to MIT's ⊃ and ASCII DC1 ◁
#define plus_plus °13    ▷ '++'; corresponds to MIT's ↑ and ASCII VT aka '\v' ◁
#define minus_minus °1    ▷ '--'; corresponds to MIT's ↓ and ASCII SOH ◁
#define minus_gt °31    ▷ '->'; corresponds to MIT's → and ASCII EM ◁
#define non_eq °32    ▷ '!='; corresponds to MIT's ≠ and ASCII SUB ◁
#define lt_eq °34    ▷ '<='; corresponds to MIT's ≤ and ASCII FS ◁
#define gt_eq °35    ▷ '>='; corresponds to MIT's ≥ and ASCII GS ◁
#define eq_eq °36    ▷ '=='; corresponds to MIT's ≡ and ASCII RS ◁
#define or_or °37    ▷ '||'; corresponds to MIT's ∨ and ASCII US ◁
#define dot_dot_dot °16    ▷ '...'; corresponds to MIT's ω and ASCII SO ◁
#define colon_colon °6    ▷ '::'; corresponds to MIT's ∈ and ASCII ACK ◁
#define period_ast °26    ▷ '.*'; corresponds to MIT's ⊗ and ASCII SYN ◁
#define minus_gt_ast °27    ▷ '->*'; corresponds to MIT's ↳ and ASCII ETB ◁
#define compress(c) if (loc++ ≤ limit) return c
⟨ Common code for CWEAVE and CTANGLE 3* ⟩ +≡
  extern char section_text[];    ▷ text being sought for ◁
  extern char *section_text_end;    ▷ end of section_text ◁
  extern char *id_first;    ▷ where the current identifier begins in the buffer ◁
  extern char *id_loc;    ▷ just after the current identifier in the buffer ◁
```

7* Code related to input routines:

```
#define xisalpha(c) (isalpha((int)(c)) ∧ ¬ishigh(c))
#define xisdigit(c) (isdigit((int)(c)) ∧ ¬ishigh(c))
#define xisspace(c) (isspace((int)(c)) ∧ ¬ishigh(c))
#define xislower(c) (islower((int)(c)) ∧ ¬ishigh(c))
#define xisupper(c) (isupper((int)(c)) ∧ ¬ishigh(c))
#define xisxdigit(c) (isxdigit((int)(c)) ∧ ¬ishigh(c))
#define isxalpha(c) ((c) ≡ '_' ∨ (c) ≡ '$')     ▷ non-alpha characters allowed in identifier ◁
#define ishigh(c) ((eight_bits)(c) > °177)
```

⟨ Common code for CWEAVE and CTANGLE 3* ⟩ +≡

```
  extern char buffer[];     ▷ where each line of input goes ◁
  extern char *buffer_end;     ▷ end of buffer ◁
  extern char *loc;     ▷ points to the next character to be read from the buffer ◁
  extern char *limit;     ▷ points to the last character in the buffer ◁
```

8* Code related to file handling:

```
  format line x     ▷ make line an unreserved word ◁
#define max_include_depth 10
          ▷ maximum number of source files open simultaneously, not counting the change file ◁
#define max_file_name_length 1024
#define cur_file file[include_depth]     ▷ current file ◁
#define cur_file_name file_name[include_depth]     ▷ current file name ◁
#define cur_line line[include_depth]     ▷ number of current line in current file ◁
#define web_file file[0]     ▷ main source file ◁
#define web_file_name file_name[0]     ▷ main source file name ◁
```

⟨ Common code for CWEAVE and CTANGLE 3* ⟩ +≡

```
  extern int include_depth;     ▷ current level of nesting ◁
  extern FILE *file[];     ▷ stack of non-change files ◁
  extern FILE *change_file;     ▷ change file ◁
  extern char file_name[][max_file_name_length];     ▷ stack of non-change file names ◁
  extern char change_file_name[];     ▷ name of change file ◁
  extern char *found_filename;     ▷ filename found by kpse_find_file ◁
  extern int line[];     ▷ number of current line in the stacked files ◁
  extern int change_line;     ▷ number of current line in change file ◁
  extern int change_depth;     ▷ where @y originated during a change ◁
  extern bool input_has_ended;     ▷ if there is no more input ◁
  extern bool changing;     ▷ if the current line is from change_file ◁
  extern bool web_file_open;     ▷ if the web file is being read ◁
```

9* ⟨ Predeclaration of procedures 4* ⟩ +≡

```
  extern bool get_line(void);     ▷ inputs the next line ◁
  extern void check_complete(void);     ▷ checks that all changes were picked up ◁
  extern void reset_input(void);     ▷ initialize to read the web file and change file ◁
```

10* Code related to section numbers:

⟨ Common code for CWEAVE and CTANGLE 3* ⟩ +≡

```
  extern sixteen_bits section_count;     ▷ the current section number ◁
  extern bool changed_section[];     ▷ is the section changed? ◁
  extern bool change_pending;     ▷ is a decision about change still unclear? ◁
  extern bool print_where;     ▷ tells CTANGLE to print line and file info ◁
```

11.* Code related to identifier and section name storage:

```
#define length(c) (size_t)((c+1)→byte_start − (c)→byte_start)  ▷ the length of a name ◁
#define print_id(c) term_write((c)→byte_start, length(c))  ▷ print identifier ◁
#define llink link  ▷ left link in binary search tree for section names ◁
#define rlink dummy.Rlink  ▷ right link in binary search tree for section names ◁
#define root name_dir→rlink  ▷ the root of the binary search tree for section names ◁
#define ilk dummy.Ilk  ▷ used by CWEAVE only ◁
⟨ Common code for CWEAVE and CTANGLE 3* ⟩ +≡
  typedef struct name_info {
    char *byte_start;  ▷ beginning of the name in byte_mem ◁
    struct name_info *link;
    union {
      struct name_info *Rlink;  ▷ right link in binary search tree for section names ◁
      eight_bits Ilk;  ▷ used by identifiers in CWEAVE only ◁
    } dummy;
    void *equiv_or_xref;  ▷ info corresponding to names ◁
  } name_info;  ▷ contains information about an identifier or section name ◁
  typedef name_info *name_pointer;  ▷ pointer into array of name_infos ◁
  typedef name_pointer *hash_pointer;
  extern char byte_mem[];  ▷ characters of names ◁
  extern char *byte_mem_end;  ▷ end of byte_mem ◁
  extern char *byte_ptr;  ▷ first unused position in byte_mem ◁
  extern name_info name_dir[];  ▷ information about names ◁
  extern name_pointer name_dir_end;  ▷ end of name_dir ◁
  extern name_pointer name_ptr;  ▷ first unused position in name_dir ◁
  extern name_pointer hash[];  ▷ heads of hash lists ◁
  extern hash_pointer hash_end;  ▷ end of hash ◁
  extern hash_pointer hash_ptr;  ▷ index into hash-head array ◁
```

12.* ⟨ Predeclaration of procedures 4* ⟩ +≡

```
  extern name_pointer id_lookup(const char *, const char *, eight_bits);
    ▷ looks up a string in the identifier table ◁
  extern name_pointer section_lookup(char *, char *, bool);  ▷ finds section name ◁
  extern void print_prefix_name(name_pointer);
  extern void print_section_name(name_pointer);
  extern void sprint_section_name(char *, name_pointer);
  extern bool names_match(name_pointer, const char *, size_t, eight_bits);
    ▷ two routines defined in ctangle.w and cweave.w ◁
  extern void init_node(name_pointer);
```

13.* Code related to error handling:

```
#define spotless 0  ▷ history value for normal jobs ◁
#define harmless_message 1  ▷ history value when non-serious info was printed ◁
#define error_message 2  ▷ history value when an error was noted ◁
#define fatal_message 3  ▷ history value when we had to stop prematurely ◁
#define mark_harmless() if (history ≡ spotless) history ← harmless_message
#define mark_error() history ← error_message
#define confusion(s) fatal(_("! This can't happen: "), s)
⟨ Common code for CWEAVE and CTANGLE 3* ⟩ +≡
  extern int history;  ▷ indicates how bad this run was ◁
```

14.* ⟨ Predeclaration of procedures 4* ⟩ +≡

```
extern int wrap_up(void);  ▷ indicate history and exit ◁
extern void err_print(const char *);  ▷ print error message and context ◁
extern void fatal(const char *, const char *);  ▷ issue error message and die ◁
extern void overflow(const char *);  ▷ succumb because a table has overflowed ◁
extern void cb_show_banner(void);  ▷ copy banner back to common.w ◁
extern void print_stats(void);  ▷ defined in ctangle.w and cweave.w ◁
```

15.* Code related to command line arguments:

```
#define show_banner flags['b']  ▷ should the banner line be printed? ◁
#define show_progress flags['p']  ▷ should progress reports be printed? ◁
#define show_happiness flags['h']  ▷ should lack of errors be announced? ◁
#define show_stats flags['s']  ▷ should statistics be printed at end of run? ◁
#define make_xrefs flags['x']  ▷ should cross references be output? ◁
#define check_for_change flags['c']  ▷ check temporary output for changes ◁
```

⟨ Common code for CWEAVE and CTANGLE 3* ⟩ +≡

```
extern int argc;  ▷ copy of ac parameter to main ◁
extern char **argv;  ▷ copy of av parameter to main ◁
extern char C_file_name[];  ▷ name of C_file ◁
extern char tex_file_name[];  ▷ name of tex_file ◁
extern char idx_file_name[];  ▷ name of idx_file ◁
extern char scn_file_name[];  ▷ name of scn_file ◁
extern char check_file_name[];  ▷ name of check_file ◁
extern bool flags[];  ▷ an option for each 7-bit code ◁
extern const char *use_language;  ▷ prefix to cwebmac.tex in TEX output ◁
```

16.* Code related to output:

```
#define update_terminal() fflush(stdout)  ▷ empty the terminal output buffer ◁
#define new_line() putchar('\n')
#define term_write(a,b) fflush(stdout), fwrite(a, sizeof(char), b, stdout)
```

⟨ Common code for CWEAVE and CTANGLE 3* ⟩ +≡

```
extern FILE *C_file;  ▷ where output of CTANGLE goes ◁
extern FILE *tex_file;  ▷ where output of CWEAVE goes ◁
extern FILE *idx_file;  ▷ where index from CWEAVE goes ◁
extern FILE *scn_file;  ▷ where list of sections from CWEAVE goes ◁
extern FILE *active_file;  ▷ currently active file for CWEAVE output ◁
extern FILE *check_file;  ▷ temporary output file ◁
```

17.* The following parameters are sufficient to handle TEX (converted to CWEB), so they should be sufficient for most applications of CWEB.

```
#define buf_size 1000  ▷ maximum length of input line, plus one ◁
#define longest_name 10000  ▷ file names, section names, and section texts shouldn't be longer than this ◁
#define long_buf_size (buf_size + longest_name)  ▷ for CWEAVE ◁
#define max_bytes 1000000
        ▷ the number of bytes in identifiers, index entries, and section names; must be less than 2^24 ◁
#define max_names 10239  ▷ number of identifiers, strings, section names; must be less than 10240 ◁
#define max_sections 4000  ▷ greater than the total number of sections ◁
```

18.* End of COMMON interface.

20*. #**define** *max_texts* 10239 ▷ number of replacement texts, must be less than 10240 ◁
#**define** *max_toks* 1000000 ▷ number of bytes in compressed C code ◁

⟨ Private variables 20* ⟩ ≡
```
static text text_info[max_texts];
static text_pointer text_info_end ← text_info + max_texts − 1;
static text_pointer text_ptr; ▷ first unused position in text_info ◁
static eight_bits tok_mem[max_toks];
static eight_bits *tok_mem_end ← tok_mem + max_toks − 1;
static eight_bits *tok_ptr; ▷ first unused position in tok_mem ◁
```
See also sections 26, 32, 38, 42, 45, 53, 57, 62, 66, 68, and 82*.

This code is used in section 1*.

29*. The following procedure is used to enter a two-byte value into *tok_mem* when a replacement text is being generated.

```
static void store_two_bytes(sixteen_bits x)
{
  if (tok_ptr + 2 > tok_mem_end) overflow(_("token"));
  *tok_ptr++ ← x ≫ 8; ▷ store high byte ◁
  *tok_ptr++ ← x & °377; ▷ store low byte ◁
}
```

35*. When the replacement text for name p is to be inserted into the output, the following subroutine is called to save the old level of output and get the new one going.

We assume that the C compiler can copy structures.

```
static void push_level( ▷ suspends the current level ◁
    name_pointer p)
{
  if (stack_ptr ≡ stack_end) overflow(_("stack"));
  *stack_ptr ← cur_state; stack_ptr++;
  if (p ≠ Λ) { ▷ p ≡ Λ means we are in output_defs ◁
    cur_name ← p; cur_repl ← (text_pointer) p→equiv; cur_byte ← cur_repl→tok_start;
    cur_section ← 0;
  }
}
```

40*. The user may have forgotten to give any C text for a section name, or the C text may have been associated with a different name by mistake.

⟨ Expand section $a - °24000$, **goto** *restart* 40* ⟩ ≡
```
{
  a −= °24000;
  if ((a + name_dir)→equiv ≠ (void *) text_info) push_level(a + name_dir);
  else if (a ≠ 0) {
    printf("%s", _("\n! Not present: <"));  print_section_name(a + name_dir);  err_print(">");
  }
  goto restart;
}
```
This code is used in section 39.

47* ⟨ If it's not there, add *cur_section_name* to the output file stack, or complain we're out of room 47* ⟩ ≡

```
{
  for (an_output_file ← cur_out_file; an_output_file < end_output_files; an_output_file++)
    if (*an_output_file ≡ cur_section_name) break;
  if (an_output_file ≡ end_output_files) {
    if (cur_out_file > output_files) *−−cur_out_file ← cur_section_name;
    else overflow(_("output␣files"));
  }
}
```

This code is used in section 77.

48* The big output switch. Here then is the routine that does the output.

```
static void phase_two(void)
{
  phase ← 2; web_file_open ← false; cur_line ← 1; ⟨ Initialize the output stacks 33 ⟩
  ⟨ Output macro definitions if appropriate 52 ⟩
  if (text_info→text_link ≡ macro ∧ cur_out_file ≡ end_output_files) {
    printf("%s", _("\n! No program text was specified.")); mark_harmless();
  }
  else {
    if (show_progress) {
      printf(cur_out_file ≡ end_output_files ? _("\nWriting the output file (%s):") :
          _("\nWriting the output files: (%s)"), C_file_name); update_terminal();
    }
    if (text_info→text_link ≠ macro) ⟨ Output material from stack 51 ⟩
    ⟨ Write all the named output files 50* ⟩
    if (show_happiness) {
      if (show_progress) new_line();
      printf("%s", _("Done."));
    }
  }
}
```

50* To write the named output files, we proceed as for the unnamed section. The only subtlety is that we have to open each one.

```
⟨ Write all the named output files 50* ⟩ ≡
  if (check_for_change) {
    fclose(C_file); C_file ← Λ; ⟨ Update the primary result when it has changed 106* ⟩
  }
  for (an_output_file ← end_output_files; an_output_file > cur_out_file; ) {
    an_output_file −−; sprint_section_name(output_file_name, *an_output_file);
    if (check_for_change) ⟨ Open the intermediate output file 105* ⟩
    else {
      fclose(C_file);
      if ((C_file ← fopen(output_file_name, "wb")) ≡ Λ)
        fatal(_("! Cannot open output file "), output_file_name);
    }
    if (show_progress) {
      printf("\n(%s)", output_file_name); update_terminal();
    }
    cur_line ← 1; ⟨ Initialize the secondary output 34 ⟩
    ⟨ Output material from stack 51 ⟩
    if (check_for_change) {
      fclose(C_file); C_file ← Λ; ⟨ Update the secondary results when they have changed 110* ⟩
    }
  }
  if (check_for_change) strcpy(check_file_name, ""); ▷ We want to get rid of the temporary file ◁
```

This code is used in section 48*.

54:* #**define** *C_printf*(*c*, *a*) *fprintf*(*C_file*, *c*, *a*)
#**define** *C_putc*(*c*) *fputc*((**int**)(*c*), *C_file*) ▷ isn't C wonderfully consistent? ◁

```
static void output_defs(void)
{
  sixteen_bits a;
  eight_bits *macro_end;
  push_level(Λ);
  for (cur_text ← text_info + 1; cur_text < text_ptr; cur_text++)
    if (cur_text→text_link ≡ macro) {      ▷ cur_text is the text for a macro ◁
      cur_byte ← cur_text→tok_start; macro_end ← (cur_text + 1)→tok_start;
        ▷ end of macro replacement text ◁
      C_printf("%s", "#define␣"); out_state ← normal; protect ← true;
        ▷ newlines should be preceded by '\\' ◁
      do macro_end−−; while (isspace(*macro_end) ∧ plus_plus ≠ *macro_end);
        ▷ discard trailing whitespace; plus_plus ≡ '\v' ◁
      while (cur_byte ≤ macro_end) {
        a ← *cur_byte++;
        if (out_state ≡ verbatim ∧ a ≠ string ∧ a ≠ constant ∧ a ≠ '\n') C_putc(a);
            ▷ a high-bit character can occur in a string ◁
        else if (a < °200) out_char(a);      ▷ one-byte token ◁
        else {
          a ← (a − °200) * °400 + *cur_byte++;
          if (a < °24000) {      ▷ °24000 ≡ (°250 − °200) * °400 ◁
            cur_val ← (int) a; out_char(identifier);
          }
          else if (a < °50000) confusion(_("macro␣defs␣have␣strange␣char"));
          else {
            cur_val ← (int) a − °50000; cur_section ← (sixteen_bits) cur_val;
            out_char(section_number);
          }      ▷ no other cases ◁
        }
      }
      protect ← false; flush_buffer( );
    }
  pop_level(false);
}
```

59:* Nowadays, most computer files are encoded in some form of "Unicode". A very convenient special case is "UTF-8", a variable-length multi-byte encoding. In order to avoid major surgery for the transliteration feature—as tempting as the extended notation `@l c3bc ue` might be—, `CTANGLE` accepts the `+u` option to activate a "poor man's UTF-8" mechanism. The first in a sequence of up to four high-bit bytes (amounting to more than 2^{20} possible character representations) determines the number of bytes used to represent the next character. Instead of extending the *translit* table to this multi-byte scenario, we simply strip all but the last byte and use this as the transliteration index.

Example: While in "classic ASCII" the German word *grün* could be treated with transliteration `@l fc ue` (from codepage ISO/IEC 8859-1) to get `gruen` as suggested above, in UTF-8 you'd be advised to use `@l bc ue` instead, because character *ü* (latin small letter u with diaeresis) is encoded as the two-byte sequence `c3 bc`, indicated by the initial three bits of byte `c3` (`1100 0011`). Note that this simple approach leads to the collision with character ¼ (vulgar fraction one quarter) with its two-byte encoding `c2 bc`.

```
#define transliterate_utf_eight flags['u']
⟨ Case of an identifier 59* ⟩ ≡
  case identifier:
    if (out_state ≡ num_or_id) C_putc('␣');
    for (j ← (cur_val + name_dir)→byte_start; j < (cur_val + name_dir + 1)→byte_start; j++)
      if (ishigh(*j)) {
        if (transliterate_utf_eight) {
          if ((eight_bits)(*j) ≥ °360) j += 3;
          else if ((eight_bits)(*j) ≥ °340) j += 2;
          else if ((eight_bits)(*j) ≥ °300) j += 1;
        }
        C_printf("%s", translit[(eight_bits)(*j) − °200]);
      }
      else C_putc(*j);
    out_state ← num_or_id; break;
```

This code is used in section 55.

67* **static bool** *skip_comment*(▷ skips over comments ◁
bool *is_long_comment*)

```
{
  char c;  ▷ current character ◁
  while (true) {
    if (loc > limit) {
      if (is_long_comment) {
        if (get_line()) return comment_continues ← true;
        else {
          err_print(_("! Input ended in mid-comment")); return comment_continues ← false;
        }
      }
      else return comment_continues ← false;
    }
    c ← *(loc++);
    if (is_long_comment ∧ c ≡ '*' ∧ *loc ≡ '/') {
      loc++; return comment_continues ← false;
    }
    if (c ≡ '@') {
      if (ccode[(eight_bits) *loc] ≡ new_section) {
        err_print(_("! Section name ended in mid-comment")); loc--;
        return comment_continues ← false;
      }
      else loc++;
    }
  }
}
```

74:* C strings and character constants, delimited by double and single quotes, respectively, can contain newlines or instances of their own delimiters if they are protected by a backslash. We follow this convention, but do not allow the string to be longer than *longest_name*.

```
⟨ Get a string 74* ⟩ ≡
  {
    char delim ← (char) c;      ▷ what started the string ◁
    id_first ← section_text + 1; id_loc ← section_text; *++id_loc ← delim;
    if (delim ≡ 'L' ∨ delim ≡ 'u' ∨ delim ≡ 'U') {      ▷ wide character constant ◁
      if (delim ≡ 'u' ∧ *loc ≡ '8') *++id_loc ← *loc++;
      delim ← *loc++; *++id_loc ← delim;
    }
    while (true) {
      if (loc ≥ limit) {
        if (*(limit − 1) ≠ '\\') {
          err_print(_("! String didn't end")); loc ← limit; break;
        }
        if (get_line() ≡ false) {
          err_print(_("! Input ended in middle of string")); loc ← buffer; break;
        }
        else if (++id_loc ≤ section_text_end) *id_loc ← '\n';      ▷ will print as "\\\n" ◁
      }
      if ((c ← (eight_bits) *loc++) ≡ delim) {
        if (++id_loc ≤ section_text_end) *id_loc ← (char) c;
        break;
      }
      if (c ≡ '\\') {
        if (loc ≥ limit) continue;
        if (++id_loc ≤ section_text_end) *id_loc ← '\\';
        c ← (eight_bits) *loc++;
      }
      if (++id_loc ≤ section_text_end) *id_loc ← (char) c;
    }
    if (id_loc ≥ section_text_end) {
      printf("%s", _("\n! String too long: ")); term_write(section_text + 1, 25); err_print("...");
    }
    id_loc++; return string;
  }
```

This code is used in section 69.

75:* After an `@` sign has been scanned, the next character tells us whether there is more work to do.

```
⟨ Get control code and possible section name 75* ⟩ ≡
  switch (c ← ccode[(eight_bits) *loc++]) {
  case ignore: continue;
  case translit_code: err_print(_("! Use @l in limbo only")); continue;
  case control_text:
    while ((c ← skip_ahead()) ≡ '@') ;      ▷ only @@ and @> are expected ◁
    if (*(loc − 1) ≠ '>') err_print(_("! Double @ should be used in control text"));
    continue;
  case section_name: cur_section_name_char ← *(loc − 1);
    ⟨ Scan the section name and make cur_section_name point to it 77 ⟩
  case string: ⟨ Scan a verbatim string 81* ⟩
  case ord: ⟨ Scan an ASCII constant 76* ⟩
  default: return c;
  }
```

This code is cited in section 92.

This code is used in section 69.

76:* After scanning a valid ASCII constant that follows `@'`, this code plows ahead until it finds the next single quote. (Special care is taken if the quote is part of the constant.) Anything after a valid ASCII constant is ignored; thus, `@'\nopq'` gives the same result as `@'\n'`.

```
⟨ Scan an ASCII constant 76* ⟩ ≡
  id_first ← loc;
  if (*loc ≡ '\\')
    if (*++loc ≡ '\'') loc++;
  while (*loc ≠ '\'') {
    if (*loc ≡ '@') {
      if (*(loc + 1) ≠ '@') err_print(_("! Double @ should be used in ASCII constant"));
      else loc++;
    }
    loc++;
    if (loc > limit) {
      err_print(_("! String didn't end")); loc ← limit − 1; break;
    }
  }
  loc++; return ord;
```

This code is used in section 75*.

79*. ⟨ Put section name into *section_text* 79* ⟩ ≡

```
while (true) {
  if (loc > limit ∧ get_line() ≡ false) {
    err_print(_("! Input ended in section name")); loc ← buffer + 1; break;
  }
  c ← (eight_bits) ∗loc; ⟨ If end of name or erroneous nesting, break 80* ⟩
  loc++;
  if (k < section_text_end) k++;
  if (xisspace(c)) {
    c ← (eight_bits) ' ';
    if (∗(k − 1) ≡ ' ') k−−;
  }
  ∗k ← (char) c;
}
if (k ≥ section_text_end) {
  printf("%s", _("\n! Section name too long: ")); term_write(section_text + 1, 25); printf("...");
  mark_harmless();
}
if (∗k ≡ ' ' ∧ k > section_text) k−−;
```

This code is used in section 77.

80*. ⟨ If end of name or erroneous nesting, **break** 80* ⟩ ≡

```
if (c ≡ '@') {
  c ← (eight_bits) ∗(loc + 1);
  if (c ≡ '>') {
    loc += 2; break;
  }
  if (ccode[(eight_bits) c] ≡ new_section) {
    err_print(_("! Section name didn't end")); break;
  }
  if (ccode[(eight_bits) c] ≡ section_name) {
    err_print(_("! Nesting of section names not allowed")); break;
  }
  ∗(++k) ← '@'; loc++;      ▷ now c ≡ ∗loc again ◁
}
```

This code is used in section 79*.

81*. At the present point in the program we have ∗(*loc* − 1) ≡ *string*; we set *id_first* to the beginning of the string itself, and *id_loc* to its ending-plus-one location in the buffer. We also set *loc* to the position just after the ending delimiter.

⟨ Scan a verbatim string 81* ⟩ ≡

```
id_first ← loc++; ∗(limit + 1) ← '@'; ∗(limit + 2) ← '>';
while (∗loc ≠ '@' ∨ ∗(loc + 1) ≠ '>') loc++;
if (loc ≥ limit) err_print(_("! Verbatim string didn't end"));
id_loc ← loc; loc += 2; return string;
```

This code is used in section 75*.

82.* Scanning a macro definition. The rules for generating the replacement texts corresponding to macros and C texts of a section are almost identical; the only differences are that

a) Section names are not allowed in macros; in fact, the appearance of a section name terminates such macros and denotes the name of the current section.
b) The symbols `@d` and `@f` and `@c` are not allowed after section names, while they terminate macro definitions.
c) Spaces are inserted after right parentheses in macros, because the ANSI C preprocessor sometimes requires it.

Therefore there is a single procedure *scan_repl* whose parameter t specifies either *macro* or *section_name*. After *scan_repl* has acted, *cur_text* will point to the replacement text just generated, and *next_control* will contain the control code that terminated the activity.

```
#define app_repl(c)
        {
          if (tok_ptr ≡ tok_mem_end) overflow(_("token"));
          else *(tok_ptr++) ← (eight_bits) c;
        }
⟨ Private variables 20* ⟩ +≡
  static text_pointer cur_text;      ▷ replacement text formed by scan_repl ◁
  static eight_bits next_control;

83.* static void scan_repl(      ▷ creates a replacement text ◁
       eight_bits t)
  {
    bool first_bracket ← true;      ▷ for cleaner output ◁
    sixteen_bits a;      ▷ the current token ◁
    if (t ≡ section_name) ⟨ Insert the line number into tok_mem 85 ⟩
    while (true)
      switch (a ← get_next()) {
      ⟨ In cases that a is a non-char token (identifier, section_name, etc.), either process it and change a
          to a byte that should be stored, or continue if a should be ignored, or goto done if a signals
          the end of this replacement text 86* ⟩
      case ')': app_repl(a);
        if (t ≡ macro ∧ first_bracket) {
          app_repl('␣'); first_bracket ← false;
        }
        break;
      default: app_repl(a);      ▷ store a in tok_mem ◁
      }
  done: next_control ← (eight_bits) a;
    if (text_ptr > text_info_end) overflow(_("text"));
    cur_text ← text_ptr; (++text_ptr)→tok_start ← tok_ptr;
  }
```

86.* ⟨ In cases that a is a non-**char** token (*identifier*, *section_name*, etc.), either process it and change a to a byte that should be stored, or **continue** if a should be ignored, or **goto** *done* if a signals the end of this replacement text 86* ⟩ ≡

```
case identifier: store_id(a);
  if (*buffer ≡ '#' ∧ ((id_loc − id_first ≡ 5 ∧ strncmp("endif", id_first, 5) ≡ 0) ∨
      (id_loc − id_first ≡ 4 ∧ strncmp("else", id_first, 4) ≡ 0) ∨
      (id_loc − id_first ≡ 4 ∧ strncmp("elif", id_first, 4) ≡ 0)))   ▷ Avoid preprocessor calamities ◁
    print_where ← true;
  break;
case section_name:
  if (t ≠ section_name) goto done;
  else {
    ⟨ Was an '@' missed here? 87* ⟩
    a ← cur_section_name − name_dir; app_repl((a/°400) + °250); app_repl(a % °400);
    ⟨ Insert the line number into tok_mem 85 ⟩
  }
  break;
case output_defs_code:
  if (t ≠ section_name) err_print(_("! Misplaced @h"));
  else {
    output_defs_seen ← true; a ← output_defs_flag; app_repl((a/°400) + °200); app_repl(a % °400);
    ⟨ Insert the line number into tok_mem 85 ⟩
  }
  break;
case constant: case string: ⟨ Copy a string or verbatim construction or numerical constant 88* ⟩
  break;
case ord: ⟨ Copy an ASCII constant 89* ⟩
  break;
case definition: case format_code: case begin_C:
  if (t ≠ section_name) goto done;
  else {
    err_print(_("! @d, @f and @c are ignored in C text")); continue;
  }
case new_section: goto done;
```

This code is used in section 83*.

87.* ⟨ Was an '@' missed here? 87* ⟩ ≡

```
{
  char *try_loc ← loc;
  while (*try_loc ≡ ' ' ∧ try_loc < limit) try_loc++;
  if (*try_loc ≡ '+' ∧ try_loc < limit) try_loc++;
  while (*try_loc ≡ ' ' ∧ try_loc < limit) try_loc++;
  if (*try_loc ≡ '=') err_print(_("! Missing '@ ' before a named section"));
      ▷ user who isn't defining a section should put newline after the name, as explained in the manual ◁
}
```

This code is used in section 86*.

88.* By default, `CTANGLE` purges single-quote characters from C++-style literals, e.g., `1'000'000`, so that you can use this notation also in C code. The `+k` switch will 'keep' the single quotes in the output.

```
#define keep_digit_separators flags['k']
```

```
⟨ Copy a string or verbatim construction or numerical constant 88* ⟩ ≡
  app_repl(a);      ▷ string or constant ◁
  while (id_first < id_loc) {      ▷ simplify @@ pairs ◁
    if (*id_first ≡ '@') {
      if (*(id_first + 1) ≡ '@') id_first++;
      else err_print(_("! Double @ should be used in string"));
    }
    else if (a ≡ constant ∧ *id_first ≡ '\'' ∧ ¬keep_digit_separators) id_first++;
    app_repl(*id_first++);
  }
  app_repl(a);
```

This code is used in section 86*.

89:* This section should be rewritten on machines that don't use ASCII code internally.

```
⟨ Copy an ASCII constant 89* ⟩ ≡
  {
    int c ← (int)((eight_bits) *id_first);
    if (c ≡ '\\') {
      c ← (int)((eight_bits) *++id_first);
      if (c ≥ '0' ∧ c ≤ '7') {
        c −= '0';
        if (*(id_first + 1) ≥ '0' ∧ *(id_first + 1) ≤ '7') {
          c ← 8 * c + *(++id_first) − '0';
          if (*(id_first + 1) ≥ '0' ∧ *(id_first + 1) ≤ '7' ∧ c < 32) c ← 8 * c + *(++id_first) − '0';
        }
      }
      else
        switch (c) {
        case 't': c ← '\t'; break;
        case 'n': c ← '\n'; break;
        case 'b': c ← '\b'; break;
        case 'f': c ← '\f'; break;
        case 'v': c ← '\v'; break;
        case 'r': c ← '\r'; break;
        case 'a': c ← '\7'; break;
        case '?': c ← '?'; break;
        case 'x':
          if (xisdigit(*(id_first + 1))) c ← (int)(*(++id_first) − '0');
          else if (xisxdigit(*(id_first + 1))) {
            ++id_first; c ← toupper((int) *id_first) − 'A' + 10;
          }
          if (xisdigit(*(id_first + 1))) c ← 16 * c + (int)(*(++id_first) − '0');
          else if (xisxdigit(*(id_first + 1))) {
            ++id_first; c ← 16 * c + toupper((int) *id_first) − (int) 'A' + 10;
          }
          break;
        case '\\': c ← '\\'; break;
        case '\'': c ← '\''; break;
        case '\"': c ← '\"'; break;
        default: err_print(_("! Unrecognized escape sequence"));
        }
    }   ▷ at this point c should have been converted to its ASCII code number ◁
    app_repl(constant);
    if (c ≥ 100) app_repl((int) '0' + c/100);
    if (c ≥ 10) app_repl((int) '0' + (c/10) % 10);
    app_repl((int) '0' + c % 10); app_repl(constant);
  }
```

This code is used in section 86*.

93* ⟨ Scan a definition 93* ⟩ ≡

```
while ((next_control ← get_next()) ≡ '\n') ;    ▷ allow newline before definition ◁
if (next_control ≠ identifier) {
  err_print(_("! Definition flushed, must start with identifier")); continue;
}
store_id(a);    ▷ append the lhs ◁
if (*loc ≠ '(') {    ▷ identifier must be separated from replacement text ◁
  app_repl(string); app_repl(' '); app_repl(string);
}
scan_repl(macro); cur_text→text_link ← macro;
```

This code is used in section 90.

100* Only a small subset of the control codes is legal in limbo, so limbo processing is straightforward.

```
static void skip_limbo(void)
{
  while (true) {
    if (loc > limit ∧ get_line() ≡ false) return;
    *(limit + 1) ← '@';
    while (*loc ≠ '@') loc++;
    if (loc++ ≤ limit) {
      char c ← *loc++;
      switch (ccode[(eight_bits) c]) {
      case new_section: return;
      case translit_code: ⟨ Read in transliteration of a character 102* ⟩
        break;
      case format_code: case '@': break;
      case control_text:
        if (c ≡ 'q' ∨ c ≡ 'Q') {
          while ((c ← (char) skip_ahead()) ≡ '@') ;
          if (*(loc − 1) ≠ '>') err_print(_("! Double @ should be used in control text"));
          break;
        }
        /* otherwise fall through */
      default: err_print(_("! Double @ should be used in limbo"));
      }
    }
  }
}
```

102:* ⟨ Read in transliteration of a character 102* ⟩ ≡

```
while (xisspace(*loc) ∧ loc < limit) loc++;
loc += 3;
if (loc > limit ∨ ¬xisxdigit(*(loc − 3)) ∨ ¬xisxdigit(*(loc − 2))
        ∨ (*(loc − 3) ≥ '0' ∧ *(loc − 3) ≤ '7') ∨ ¬xisspace(*(loc − 1)))
  err_print(_("! Improper hex number following @l"));
else {
  unsigned int i;
  char *beg;
  sscanf(loc − 3, "%x", &i);
  while (xisspace(*loc) ∧ loc < limit) loc++;
  beg ← loc;
  while (loc < limit ∧ (xisalpha(*loc) ∨ xisdigit(*loc) ∨ isxalpha(*loc))) loc++;
  if (loc − beg ≥ translit_length) err_print(_("! Replacement string in @l too long"));
  else {
    strncpy(translit[i − °200], beg, (size_t)(loc − beg)); translit[i − °200][loc − beg] ← '\0';
  }
}
```

This code is used in section 100*.

103:* Because on some systems the difference between two pointers is a **ptrdiff_t** but not an **int**, we use `%td` to print these quantities.

```
void print_stats(void)
{
  puts(_("\nMemory usage statistics:"));
  printf(_("%td names (out of %ld)\n"), (ptrdiff_t)(name_ptr − name_dir), (long) max_names);
  printf(_("%td replacement texts (out of %ld)\n"), (ptrdiff_t)(text_ptr − text_info),
      (long) max_texts);
  printf(_("%td bytes (out of %ld)\n"), (ptrdiff_t)(byte_ptr − byte_mem), (long) max_bytes);
  printf(_("%td tokens (out of %ld)\n"), (ptrdiff_t)(tok_ptr − tok_mem), (long) max_toks);
}
```

104.* Extensions to CWEB. The following sections introduce new or improved features that have been created by numerous contributors over the course of a quarter century.

Care has been taken to keep the original section numbering intact, so this new material should nicely integrate with the original "**104. Index**."

105* Output file update. Most C projects are controlled by a `Makefile` that automatically takes care of the temporal dependencies between the different source modules. It may be convenient that `CWEB` doesn't create new output for all existing files, when there are only changes to some of them. Thus the `make` process will only recompile those modules where necessary. You can activate this feature with the '`+c`' command-line option. The idea and basic implementation of this mechanism can be found in the program `NUWEB` by Preston Briggs, to whom credit is due.

```
⟨ Open the intermediate output file 105* ⟩ ≡
  {
    if ((C_file ← fopen(output_file_name, "a")) ≡ Λ)
      fatal(_("! Cannot open output file "), output_file_name);
    else fclose(C_file);      ▷ Test accessability ◁
    if ((C_file ← fopen(check_file_name, "wb")) ≡ Λ)
      fatal(_("! Cannot open output file "), check_file_name);
  }
```

This code is used in section 50*.

106* ⟨ Update the primary result when it has changed 106* ⟩ ≡

```
if ((C_file ← fopen(C_file_name, "r")) ≠ Λ) {
  ⟨ Set up the comparison of temporary output 107* ⟩
  ⟨ Create the primary output depending on the comparison 109* ⟩
}
else rename(check_file_name, C_file_name);      ▷ This was the first run ◁
```

This code is used in section 50*.

107* ⟨ Set up the comparison of temporary output 107* ⟩ ≡

```
bool comparison ← false;
if ((check_file ← fopen(check_file_name, "r")) ≡ Λ)
  fatal(_("! Cannot open output file "), check_file_name);
⟨ Compare the temporary output to the previous output 108* ⟩
fclose(C_file); C_file ← Λ; fclose(check_file); check_file ← Λ;
```

This code is used in sections 106* and 110*.

108* We hope that this runs fast on most systems.

```
⟨ Compare the temporary output to the previous output 108* ⟩ ≡
  do {
    char x[BUFSIZ], y[BUFSIZ];
    int x_size ← fread(x, sizeof(char), BUFSIZ, C_file);
    int y_size ← fread(y, sizeof(char), BUFSIZ, check_file);
    comparison ← (x_size ≡ y_size) ∧ ¬memcmp(x, y, x_size);
  } while (comparison ∧ ¬feof(C_file) ∧ ¬feof(check_file));
```

This code is used in section 107*.

109* Note the superfluous call to *remove* before *rename*. We're using it to get around a bug in some implementations of *rename*.

```
⟨ Create the primary output depending on the comparison 109* ⟩ ≡
  if (comparison) remove(check_file_name);      ▷ The output remains untouched ◁
  else {
    remove(C_file_name); rename(check_file_name, C_file_name);
  }
```

This code is used in section 106*.

110*. The author of a `CWEB` program may want to write the *secondary* output instead of to a file (in `@(...@>`) to `/dev/null` or `/dev/stdout` or `/dev/stderr`. We must take care of the *temporary* output already written to a file and finally get rid of that file.

```
⟨ Update the secondary results when they have changed 110* ⟩ ≡
  if (0 ≡ strcmp("/dev/stdout", output_file_name))
    ⟨ Redirect temporary output to /dev/stdout 112* ⟩
  else if (0 ≡ strcmp("/dev/stderr", output_file_name))
    ⟨ Redirect temporary output to /dev/stderr 113* ⟩
  else if (0 ≡ strcmp("/dev/null", output_file_name))
    ⟨ Redirect temporary output to /dev/null 114* ⟩
  else {      ▷ Hopefully a regular output file ◁
    if ((C_file ← fopen(output_file_name, "r")) ≠ Λ) {
      ⟨ Set up the comparison of temporary output 107* ⟩
      ⟨ Create the secondary output depending on the comparison 111* ⟩
    }
    else rename(check_file_name, output_file_name);      ▷ This was the first run ◁
  }
```

This code is used in section 50*.

111*. Again, we use a call to *remove* before *rename*.

```
⟨ Create the secondary output depending on the comparison 111* ⟩ ≡
  if (comparison) remove(check_file_name);      ▷ The output remains untouched ◁
  else {
    remove(output_file_name); rename(check_file_name, output_file_name);
  }
```

This code is used in sections 110*, 112*, 113*, and 114*.

112*. Copy secondary output to *stdout*.

```
⟨ Redirect temporary output to /dev/stdout 112* ⟩ ≡
  {
    ⟨ Setup system redirection 115* ⟩
    do {
      in_size ← fread(in_buf, sizeof(char), BUFSIZ, check_file); in_buf[in_size] ← '\0';
      fprintf(stdout, "%s", in_buf);
    } while (¬feof(check_file));
    fclose(check_file); check_file ← Λ; ⟨ Create the secondary output depending on the comparison 111* ⟩
  }
```

This code is used in section 110*.

113*. Copy secondary output to *stderr*.

```
⟨ Redirect temporary output to /dev/stderr 113* ⟩ ≡
  {
    ⟨ Setup system redirection 115* ⟩
    do {
      in_size ← fread(in_buf, sizeof(char), BUFSIZ, check_file); in_buf[in_size] ← '\0';
      fprintf(stderr, "%s", in_buf);
    } while (¬feof(check_file));
    fclose(check_file); check_file ← Λ; ⟨ Create the secondary output depending on the comparison 111* ⟩
  }
```

This code is used in section 110*.

114.* No copying necessary, just remove the temporary output file.

```
⟨ Redirect temporary output to /dev/null 114* ⟩ ≡
  {
    bool comparison ← true;
    ⟨ Create the secondary output depending on the comparison 111* ⟩
  }
```

This code is used in section 110*.

115.* ⟨ Setup system redirection 115* ⟩ ≡

```
char in_buf[BUFSIZ + 1];
int in_size;
bool comparison ← true;
if ((check_file ← fopen(check_file_name, "r")) ≡ Λ)
  fatal(_("! Cannot open output file "), check_file_name);
```

This code is used in sections 112* and 113*.

116.* Print "version" information. Don't do this at home, kids! Push our local macro to the variable in `COMMON` for printing the *banner* and the *versionstring* from there.

```
#define max_banner 50
⟨ Common code for CWEAVE and CTANGLE 3* ⟩ +≡
  extern char cb_banner[];
```

117.* ⟨ Set initial values 21 ⟩ +≡

```
strncpy(cb_banner, banner, max_banner − 1);
```

118.* Index. Here is a cross-reference table for CTANGLE. All sections in which an identifier is used are listed with that identifier, except that reserved words are indexed only when they appear in format definitions, and the appearances of identifiers in section names are not indexed. Underlined entries correspond to where the identifier was declared. Error messages and a few other things like "ASCII code dependencies" are indexed here too.

The following sections were changed by the change file: 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 20, 29, 35, 40, 47, 48, 50, 54, 59, 67, 74, 75, 76, 79, 80, 81, 82, 83, 86, 87, 88, 89, 93, 100, 102, 103, 104, 105, 106, 107, 108, 109, 110, 111, 112, 113, 114, 115, 116, 117, 118.

⟨ Case of a section number 60 ⟩ Used in section 55.
⟨ Case of an identifier 59* ⟩ Used in section 55.
⟨ Cases like != 56 ⟩ Used in section 55.
⟨ Common code for CWEAVE and CTANGLE 3*, 6*, 7*, 8*, 10*, 11*, 13*, 15*, 16*, 116* ⟩ Used in section 1*.
⟨ Compare the temporary output to the previous output 108* ⟩ Used in section 107*.
⟨ Compress two-symbol operator 71 ⟩ Used in section 69.
⟨ Copy a string or verbatim construction or numerical constant 88* ⟩ Used in section 86*.
⟨ Copy an ASCII constant 89* ⟩ Used in section 86*.
⟨ Create the primary output depending on the comparison 109* ⟩ Used in section 106*.
⟨ Create the secondary output depending on the comparison 111* ⟩ Used in sections 110*, 112*, 113*, and 114*.
⟨ Expand section $a - °24000$, **goto** *restart* 40* ⟩ Used in section 39.
⟨ Get a constant 73 ⟩ Used in section 69.
⟨ Get a string 74* ⟩ Used in section 69.
⟨ Get an identifier 72 ⟩ Used in section 69.
⟨ Get control code and possible section name 75* ⟩ Cited in section 92. Used in section 69.
⟨ If end of name or erroneous nesting, **break** 80* ⟩ Used in section 79*.
⟨ If it's not there, add *cur_section_name* to the output file stack, or complain we're out of room 47* ⟩ Used in section 77.
⟨ If section is not being defined, **continue** 94 ⟩ Used in section 90.
⟨ In cases that a is a non-**char** token (*identifier*, *section_name*, etc.), either process it and change a to a byte that should be stored, or **continue** if a should be ignored, or **goto** *done* if a signals the end of this replacement text 86* ⟩ Used in section 83*.
⟨ Include files 5* ⟩ Used in section 1*.
⟨ Initialize the output stacks 33 ⟩ Used in section 48*.
⟨ Initialize the secondary output 34 ⟩ Used in section 50*.
⟨ Insert the line number into *tok_mem* 85 ⟩ Used in sections 69, 83*, and 86*.
⟨ Insert the section number into *tok_mem* 96 ⟩ Used in section 95.
⟨ Open the intermediate output file 105* ⟩ Used in section 50*.
⟨ Output macro definitions if appropriate 52 ⟩ Used in section 48*.
⟨ Output material from *stack* 51 ⟩ Used in sections 48* and 50*.
⟨ Predeclaration of procedures 4*, 9*, 12*, 14*, 30, 37, 44, 49, 65, 70, 84, 91, 99, 101 ⟩ Used in section 1*.
⟨ Private variables 20*, 26, 32, 38, 42, 45, 53, 57, 62, 66, 68, 82* ⟩ Used in section 1*.
⟨ Put section name into *section_text* 79* ⟩ Used in section 77.
⟨ Read in transliteration of a character 102* ⟩ Used in section 100*.
⟨ Redirect temporary output to /dev/null 114* ⟩ Used in section 110*.
⟨ Redirect temporary output to /dev/stderr 113* ⟩ Used in section 110*.
⟨ Redirect temporary output to /dev/stdout 112* ⟩ Used in section 110*.
⟨ Scan a definition 93* ⟩ Used in section 90.
⟨ Scan a verbatim string 81* ⟩ Used in section 75*.
⟨ Scan an ASCII constant 76* ⟩ Used in section 75*.
⟨ Scan the C part of the current section 95 ⟩ Used in section 90.
⟨ Scan the section name and make *cur_section_name* point to it 77 ⟩ Used in section 75*.
⟨ Set initial values 21, 23, 27, 46, 58, 63, 78, 117* ⟩ Used in section 2*.
⟨ Set up the comparison of temporary output 107* ⟩ Used in sections 106* and 110*.
⟨ Setup system redirection 115* ⟩ Used in sections 112* and 113*.
⟨ Skip ahead until *next_control* corresponds to @d, @<, @␣ or the like 92 ⟩ Used in section 90.
⟨ Typedef declarations 19, 31 ⟩ Used in section 1*.
⟨ Update the data structure so that the replacement text is accessible 97 ⟩ Used in section 95.
⟨ Update the primary result when it has changed 106* ⟩ Used in section 50*.
⟨ Update the secondary results when they have changed 110* ⟩ Used in section 50*.
⟨ Was an '@' missed here? 87* ⟩ Used in section 86*.
⟨ Write all the named output files 50* ⟩ Used in section 48*.