The CTANGLE processor

(Version 4.7)

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␣4.7)"
```

⟨ Include files 4 ⟩
⟨ Preprocessor definitions ⟩
⟨ Common code for CWEAVE and CTANGLE 3 ⟩
⟨ Typedef declarations 19 ⟩
⟨ Private variables 20 ⟩
⟨ Predeclaration of procedures 8 ⟩

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) puts(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:

```
#define ctangle false
#define cweave true
```

⟨ Common code for CWEAVE and CTANGLE 3 ⟩ ≡

```
typedef bool boolean;
typedef uint8_t eight_bits;
typedef uint16_t sixteen_bits;
extern boolean program;      ▷ CWEAVE or CTANGLE? ◁
extern int phase;      ▷ which phase are we in? ◁
```

See also sections 5, 6, 7, 9, 10, 12, 14, and 15.

This code is used in section 1.

4. Interface to the standard C library:

```
⟨ Include files 4 ⟩ ≡
#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 ◁
```

This code is used in section 1.

5. Code related to the character set:

```
#define and_and °4      ▷ '&&'; corresponds to MIT's ∧ ◁
#define lt_lt °20       ▷ '<<'; corresponds to MIT's ⊂ ◁
#define gt_gt °21       ▷ '>>'; corresponds to MIT's ⊃ ◁
#define plus_plus °13   ▷ '++'; corresponds to MIT's ↑ ◁
#define minus_minus °1  ▷ '--'; corresponds to MIT's ↓ ◁
#define minus_gt °31    ▷ '->'; corresponds to MIT's → ◁
#define non_eq °32      ▷ '!='; corresponds to MIT's ≠ ◁
#define lt_eq °34       ▷ '<='; corresponds to MIT's ≤ ◁
#define gt_eq °35       ▷ '>='; corresponds to MIT's ≥ ◁
#define eq_eq °36       ▷ '=='; corresponds to MIT's ≡ ◁
#define or_or °37       ▷ '||'; corresponds to MIT's ∨ ◁
#define dot_dot_dot °16 ▷ '...'; corresponds to MIT's ϖ ◁
#define colon_colon °6  ▷ '::'; corresponds to MIT's ∈ ◁
#define period_ast °26  ▷ '.*'; corresponds to MIT's ⊗ ◁
#define minus_gt_ast °27 ▷ '->*'; corresponds to MIT's ↰ ◁
#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 ◁
```

6. Code related to input routines:

```
#define xisalpha(c) (isalpha((int)(c)) ∧ ((eight_bits)(c) < °200))
#define xisdigit(c) (isdigit((int)(c)) ∧ ((eight_bits)(c) < °200))
#define xisspace(c) (isspace((int)(c)) ∧ ((eight_bits)(c) < °200))
#define xislower(c) (islower((int)(c)) ∧ ((eight_bits)(c) < °200))
#define xisupper(c) (isupper((int)(c)) ∧ ((eight_bits)(c) < °200))
#define xisxdigit(c) (isxdigit((int)(c)) ∧ ((eight_bits)(c) < °200))
#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 ◁
```

7. 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 60
#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 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 boolean input_has_ended;      ▷ if there is no more input ◁
  extern boolean changing;      ▷ if the current line is from change_file ◁
  extern boolean web_file_open;      ▷ if the web file is being read ◁
```

8. ⟨ Predeclaration of procedures 8 ⟩ ≡

```
  extern boolean 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 ◁
```

See also sections 11, 13, 16, 30, 35, 39, 44, 49, 53, 65, 70, 84, 91, 99, and 101.

This code is used in section 1.

9. Code related to section numbers:

⟨ Common code for CWEAVE and CTANGLE 3 ⟩ +≡

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

10. 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 ◁
⟨ 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 ◁
      char 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 h;  ▷ index into hash-head array ◁
```

11. ⟨ Predeclaration of procedures 8 ⟩ +≡

```
  extern boolean names_match(name_pointer, const char *, size_t, eight_bits);
  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 *, boolean);  ▷ finds section name ◁
  extern void init_node(name_pointer);
  extern void init_p(name_pointer, eight_bits);
  extern void print_prefix_name(name_pointer);
  extern void print_section_name(name_pointer);
  extern void sprint_section_name(char *, name_pointer);
```

12. 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 ◁
```

13. ⟨ Predeclaration of procedures 8 ⟩ +≡
```
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 ◁
```

14. 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? ◁
```

⟨ 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 boolean flags[];  ▷ an option for each 7-bit code ◁
```

15. 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 ◁
```

16. The procedure that gets everything rolling:

⟨ Predeclaration of procedures 8 ⟩ +≡
```
extern void common_init(void);
extern void print_stats(void);
```

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 200  ▷ 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 100000
        ▷ the number of bytes in identifiers, index entries, and section names; must be less than 2^24 ◁
#define max_names 5000  ▷ number of identifiers, strings, section names; must be less than 10240 ◁
#define max_sections 2000  ▷ greater than the total number of sections ◁
```

18. End of COMMON interface.

19. Data structures exclusive to CTANGLE. We've already seen that the *byte_mem* array holds the names of identifiers, strings, and sections; the *tok_mem* array holds the replacement texts for sections. Allocation is sequential, since things are deleted only during Phase II, and only in a last-in-first-out manner.

A **text** variable is a structure containing a pointer into *tok_mem*, which tells where the corresponding text starts, and an integer *text_link*, which, as we shall see later, is used to connect pieces of text that have the same name. All the **text**s are stored in the array *text_info*, and we use a **text_pointer** variable to refer to them.

The first position of *tok_mem* that is unoccupied by replacement text is called *tok_ptr*, and the first unused location of *text_info* is called *text_ptr*. Thus we usually have the identity *text_ptr*→*tok_start* ≡ *tok_ptr*.

```
⟨ Typedef declarations 19 ⟩ ≡
  typedef struct {
    eight_bits *tok_start;      ▷ pointer into tok_mem ◁
    sixteen_bits text_link;     ▷ relates replacement texts ◁
  } text;
  typedef text *text_pointer;
```

See also section 31.

This code is used in section 1.

20. #**define** *max_texts* 4000 ▷ number of replacement texts, must be less than 10240 ◁
#**define** *max_toks* 270000 ▷ 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, 37, 42, 45, 52, 57, 62, 66, 68, and 82.

This code is used in section 1.

21.

```
⟨ Set initial values 21 ⟩ ≡
  text_info→tok_start ← tok_ptr ← tok_mem; text_ptr ← text_info + 1; text_ptr→tok_start ← tok_mem;
    ▷ this makes replacement text 0 of length zero ◁
```

See also sections 23, 27, 46, 58, 63, and 78.

This code is used in section 2.

22. If *p* is a pointer to a section name, *p*→*equiv* is a pointer to its replacement text, an element of the array *text_info*.

#**define** *equiv* *equiv_or_xref* ▷ info corresponding to names ◁

23.

```
⟨ Set initial values 21 ⟩ +≡
  init_node(name_dir);      ▷ the undefined section has no replacement text ◁
```

24. Here's the procedure that decides whether a name of length l starting at position *first* equals the identifier pointed to by p:

```
boolean names_match(name_pointer p,      ▷ points to the proposed match ◁
    const char *first,      ▷ position of first character of string ◁
    size_t l,      ▷ length of identifier ◁
    eight_bits t)      ▷ not used by CTANGLE ◁
{ (void) t;
  return length(p) ≡ l ∧ strncmp(first, p→byte_start, l) ≡ 0;
}
```

25. The common lookup routine refers to separate routines *init_node* and *init_p* when the data structure grows. Actually *init_p* is called only by CWEAVE, but we need to declare a dummy version so that the loader won't complain of its absence.

```
void init_node(name_pointer node)
{
  node→equiv ← (void *) text_info;
}
void init_p(name_pointer p, eight_bits t)
{ (void) p; (void) t; }
```

26. Tokens. Replacement texts, which represent C code in a compressed format, appear in *tok_mem* as mentioned above. The codes in these texts are called 'tokens'; some tokens occupy two consecutive eight-bit byte positions, and the others take just one byte.

If p points to a replacement text, $p\rightarrow tok\_start$ is the *tok_mem* position of the first eight-bit code of that text. If $p\rightarrow text\_link \equiv 0$, this is the replacement text for a macro, otherwise it is the replacement text for a section. In the latter case $p\rightarrow text\_link$ is either equal to *section_flag*, which means that there is no further text for this section, or $p\rightarrow text\_link$ points to a continuation of this replacement text; such links are created when several sections have C texts with the same name, and they also tie together all the C texts of unnamed sections. The replacement text pointer for the first unnamed section appears in $text\_info\rightarrow text\_link$, and the most recent such pointer is *last_unnamed*.

```
#define macro 0
#define section_flag max_texts      ▷ final text_link in section replacement texts ◁
⟨ Private variables 20 ⟩ +≡
  static text_pointer last_unnamed;      ▷ most recent replacement text of unnamed section ◁
```

27. ⟨ Set initial values 21 ⟩ +≡

```
last_unnamed ← text_info; text_info→text_link ← macro;
```

28. If the first byte of a token is less than $°200$, the token occupies a single byte. Otherwise we make a sixteen-bit token by combining two consecutive bytes a and b. If $°200 \le a < °250$, then $(a - °200) \times 2^8 + b$ points to an identifier; if $°250 \le a < °320$, then $(a - °250) \times 2^8 + b$ points to a section name (or, if it has the special value *output_defs_flag*, to the area where the preprocessor definitions are stored); and if $°320 \le a < °400$, then $(a - °320) \times 2^8 + b$ is the number of the section in which the current replacement text appears.

Codes less than $°200$ are 7-bit **char** codes that represent themselves. Some of the 7-bit codes will not be present, however, so we can use them for special purposes. The following symbolic names are used:

string denotes the beginning or end of a string or a verbatim construction.
constant denotes a numerical constant.
join denotes the concatenation of adjacent items with no space or line breaks allowed between them (the `@&` operation of `CWEB`).

```
#define string °2      ▷ takes the place of ASCII STX ◁
#define constant °3      ▷ takes the place of ASCII ETX ◁
#define join °177      ▷ takes the place of ASCII DEL ◁
#define output_defs_flag (2 * °24000 - 1)      ▷ °24000 ≡ (°250 - °200) * °400 ◁
```

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 ◁
}
```

30. ⟨ Predeclaration of procedures 8 ⟩ +≡ **static void** *store_two_bytes*(**sixteen_bits**);

31. Stacks for output. The output process uses a stack to keep track of what is going on at different "levels" as the sections are being written out. Entries on this stack have five parts:

end_field is the *tok_mem* location where the replacement text of a particular level will end;
byte_field is the *tok_mem* location from which the next token on a particular level will be read;
name_field points to the name corresponding to a particular level;
repl_field points to the replacement text currently being read at a particular level;
section_field is the section number, or zero if this is a macro.

The current values of these five quantities are referred to quite frequently, so they are stored in a separate place instead of in the *stack* array. We call the current values *cur_end*, *cur_byte*, *cur_name*, *cur_repl*, and *cur_section*.

The global variable *stack_ptr* tells how many levels of output are currently in progress. The end of all output occurs when the stack is empty, i.e., when *stack_ptr* ≡ *stack*.

```
⟨ Typedef declarations 19 ⟩ +≡
  typedef struct {
    eight_bits *end_field;      ▷ ending location of replacement text ◁
    eight_bits *byte_field;     ▷ present location within replacement text ◁
    name_pointer name_field;    ▷ byte_start index for text being output ◁
    text_pointer repl_field;    ▷ tok_start index for text being output ◁
    sixteen_bits section_field;     ▷ section number or zero if not a section ◁
  } output_state;
  typedef output_state *stack_pointer;
```

32.
```
#define stack_size 50      ▷ number of simultaneous levels of macro expansion ◁
#define cur_end cur_state.end_field      ▷ current ending location in tok_mem ◁
#define cur_byte cur_state.byte_field      ▷ location of next output byte in tok_mem ◁
#define cur_name cur_state.name_field      ▷ pointer to current name being expanded ◁
#define cur_repl cur_state.repl_field      ▷ pointer to current replacement text ◁
#define cur_section cur_state.section_field      ▷ current section number being expanded ◁
⟨ Private variables 20 ⟩ +≡
  static output_state cur_state;      ▷ cur_end, cur_byte, cur_name, cur_repl, and cur_section ◁
  static output_state stack[stack_size + 1];      ▷ info for non-current levels ◁
  static stack_pointer stack_end ← stack + stack_size;      ▷ end of stack ◁
  static stack_pointer stack_ptr;      ▷ first unused location in the output state stack ◁
```

33. To get the output process started, we will perform the following initialization steps. We may assume that *text_info→text_link* is nonzero, since it points to the C text in the first unnamed section that generates code; if there are no such sections, there is nothing to output, and an error message will have been generated before we do any of the initialization.

```
⟨ Initialize the output stacks 33 ⟩ ≡
  stack_ptr ← stack + 1; cur_name ← name_dir; cur_repl ← text_info→text_link + text_info;
  cur_byte ← cur_repl→tok_start; cur_end ← (cur_repl + 1)→tok_start; cur_section ← 0;
```

This code is used in section 48.

34. 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_end ← (cur_repl + 1)→tok_start; cur_section ← 0;
  }
}
```

35. ⟨ Predeclaration of procedures 8 ⟩ +≡

```
static void push_level(name_pointer);
static void pop_level(boolean);
```

36. When we come to the end of a replacement text, the *pop_level* subroutine does the right thing: It either moves to the continuation of this replacement text or returns the state to the most recently stacked level.

```
static void pop_level(      ▷ do this when cur_byte reaches cur_end ◁
    boolean flag)      ▷ flag ≡ false means we are in output_defs ◁
{
  if (flag ∧ cur_repl→text_link < section_flag) {      ▷ link to a continuation ◁
    cur_repl ← cur_repl→text_link + text_info;      ▷ stay on the same level ◁
    cur_byte ← cur_repl→tok_start; cur_end ← (cur_repl + 1)→tok_start; return;
  }
  stack_ptr--;      ▷ go down to the previous level ◁
  if (stack_ptr > stack) cur_state ← *stack_ptr;
}
```

37. The heart of the output procedure is the function *get_output*, which produces the next token of output and sends it on to the lower-level function *out_char*. The main purpose of *get_output* is to handle the necessary stacking and unstacking. It sends the value *section_number* if the next output begins or ends the replacement text of some section, in which case *cur_val* is that section's number (if beginning) or the negative of that value (if ending). (A section number of 0 indicates not the beginning or ending of a section, but a #**line** command.) And it sends the value *identifier* if the next output is an identifier, in which case *cur_val* points to that identifier name.

```
#define section_number °201      ▷ code returned by get_output for section numbers ◁
#define identifier °202      ▷ code returned by get_output for identifiers ◁
```

⟨ Private variables 20 ⟩ +≡

```
static int cur_val;      ▷ additional information corresponding to output token ◁
```

38. If *get_output* finds that no more output remains, it returns with *stack_ptr* ≡ *stack*.

```
static void get_output(void)      ▷ sends next token to out_char ◁
{
  sixteen_bits a;      ▷ value of current byte ◁
restart:
  if (stack_ptr ≡ stack) return;
  if (cur_byte ≡ cur_end) {
    cur_val ← −((int) cur_section);      ▷ cast needed because of sign extension ◁
    pop_level(true);
    if (cur_val ≡ 0) goto restart;
    out_char(section_number); return;
  }
  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++;
    switch (a/°24000) {      ▷ °24000 ≡ (°250 − °200) ∗ °400 ◁
    case 0: cur_val ← (int) a; out_char(identifier); break;
    case 1:
      if (a ≡ output_defs_flag) output_defs( );
      else ⟨ Expand section a − °24000, goto restart 40 ⟩
      break;
    default: cur_val ← (int) a − °50000;
      if (cur_val > 0) cur_section ← (sixteen_bits) cur_val;
      out_char(section_number);
    }
  }
}
```

39. ⟨ Predeclaration of procedures 8 ⟩ +≡ `static void get_output(void);`

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) {
    fputs("\n!␣Not␣present:␣<", stdout); print_section_name(a + name_dir); err_print(">");
  }
  goto restart;
}
```

This code is used in section 38.

41. Producing the output. The *get_output* routine above handles most of the complexity of output generation, but there are two further considerations that have a nontrivial effect on CTANGLE's algorithms.

42. First, we want to make sure that the output has spaces and line breaks in the right places (e.g., not in the middle of a string or a constant or an identifier, not at a '@&' position where quantities are being joined together, and certainly after an = because the C compiler thinks =- is ambiguous).

The output process can be in one of following states:

num_or_id means that the last item in the buffer is a number or identifier, hence a blank space or line break must be inserted if the next item is also a number or identifier.

unbreakable means that the last item in the buffer was followed by the @& operation that inhibits spaces between it and the next item.

verbatim means we're copying only character tokens, and that they are to be output exactly as stored. This is the case during strings, verbatim constructions and numerical constants.

post_slash means we've just output a slash.

normal means none of the above.

Furthermore, if the variable *protect* is *true*, newlines are preceded by a '\'.

```
#define normal 0 ▷ non-unusual state ◁
#define num_or_id 1 ▷ state associated with numbers and identifiers ◁
#define post_slash 2 ▷ state following a / ◁
#define unbreakable 3 ▷ state associated with @& ◁
#define verbatim 4 ▷ state in the middle of a string ◁
⟨ Private variables 20 ⟩ +≡
  static eight_bits out_state; ▷ current status of partial output ◁
  static boolean protect; ▷ should newline characters be quoted? ◁
```

43. Here is a routine that is invoked when we want to output the current line. During the output process, *cur_line* equals the number of the next line to be output.

```
static void flush_buffer(void) ▷ writes one line to output file ◁
{
  C_putc('\n');
  if (cur_line % 100 ≡ 0 ∧ show_progress) {
    putchar('.');
    if (cur_line % 500 ≡ 0) printf("%d", cur_line);
    update_terminal; ▷ progress report ◁
  }
  cur_line++;
}
```

44. ⟨ Predeclaration of procedures 8 ⟩ +≡ `static void flush_buffer(void);`

45. Second, we have modified the original TANGLE so that it will write output on multiple files. If a section name is introduced in at least one place by @(instead of @<, we treat it as the name of a file. All these special sections are saved on a stack, *output_files*. We write them out after we've done the unnamed section.

```
#define max_files 256
⟨ Private variables 20 ⟩ +≡
  static name_pointer output_files[max_files];
  static name_pointer *cur_out_file, *end_output_files, *an_output_file;
  static char cur_section_name_char; ▷ is it '<' or '(' ◁
  static char output_file_name[longest_name + 1]; ▷ name of the file ◁
```

46. We make *end_output_files* point just beyond the end of *output_files*. The stack pointer *cur_out_file* starts out there. Every time we see a new file, we decrement *cur_out_file* and then write it in.

```
⟨ Set initial values 21 ⟩ +≡
  cur_out_file ← end_output_files ← output_files + max_files;
```

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 51 ⟩
  if (text_info→text_link ≡ macro ∧ cur_out_file ≡ end_output_files) {
    fputs("\n!␣No␣program␣text␣was␣specified.", stdout); mark_harmless;
  }
  else {
    if (cur_out_file ≡ end_output_files) {
      if (show_progress) {
        printf("\nWriting␣the␣output␣file␣(%s):", C_file_name); update_terminal;
      }
    }
    else {
      if (show_progress) {
        fputs("\nWriting␣the␣output␣files:", stdout); printf("␣(%s)", C_file_name);
        update_terminal;
      }
      if (text_info→text_link ≡ macro) goto writeloop;
    }
    while (stack_ptr > stack) get_output( );
    flush_buffer( );
  writeloop: ⟨ Write all the named output files 50 ⟩
    if (show_happiness) {
      if (show_progress) new_line;
      fputs("Done.", stdout);
    }
  }
}
```

49. ⟨ Predeclaration of procedures 8 ⟩ +≡ **static void** *phase_two*(**void**);

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 ⟩ ≡
  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); 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; stack_ptr ← stack + 1; cur_name ← ∗an_output_file; cur_repl ← (text_pointer)
        cur_name→equiv; cur_byte ← cur_repl→tok_start; cur_end ← (cur_repl + 1)→tok_start;
    while (stack_ptr > stack) get_output( );
    flush_buffer( );
  }
```

This code is used in section 48.

51. If a `@h` was not encountered in the input, we go through the list of replacement texts and copy the ones that refer to macros, preceded by the `#define` preprocessor command.

```
⟨ Output macro definitions if appropriate 51 ⟩ ≡
  if (¬output_defs_seen) output_defs( );
```

This code is used in section 48.

52.
```
⟨ Private variables 20 ⟩ +≡
  static boolean output_defs_seen ← false;
```

53.
```
⟨ Predeclaration of procedures 8 ⟩ +≡
  static void output_defs(void);
  static void out_char(eight_bits);
```

54.
```
#define C_printf(c, a) fprintf(C_file, c, a)
#define C_putc(c) putc((int)(c), C_file)      ▷ isn't C wonderfully consistent? ◁
  static void output_defs(void)
  {
    sixteen_bits a;
    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; cur_end ← (cur_text + 1)→tok_start; C_printf("%s","#define␣");
        out_state ← normal; protect ← true;      ▷ newlines should be preceded by '\\' ◁
        while (cur_byte < cur_end) {
          a ← *cur_byte++;
          if (cur_byte ≡ cur_end ∧ a ≡ '\n') break;      ▷ disregard a final newline ◁
          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);
  }
```

55. A many-way switch is used to send the output. Note that this function is not called if *out_state* ≡ *verbatim*, except perhaps with arguments `'\n'` (protect the newline), *string* (end the string), or *constant* (end the constant).

```
static void out_char(eight_bits cur_char)
{
  char *j, *k;      ▷ pointer into byte_mem ◁
restart:
  switch (cur_char) {
  case '\n':
    if (protect ∧ out_state ≠ verbatim) C_putc('␣');
    if (protect ∨ out_state ≡ verbatim) C_putc('\\');
    flush_buffer( );
    if (out_state ≠ verbatim) out_state ← normal;
    break;
  ⟨ Case of an identifier 59 ⟩
  ⟨ Case of a section number 60 ⟩
  ⟨ Cases like != 56 ⟩
  case '=': case '>': C_putc(cur_char); C_putc('␣'); out_state ← normal; break;
  case join: out_state ← unbreakable; break;
  case constant:
    if (out_state ≡ verbatim) {
      out_state ← num_or_id; break;
    }
    if (out_state ≡ num_or_id) C_putc('␣');
    out_state ← verbatim; break;
  case string:
    if (out_state ≡ verbatim) out_state ← normal;
    else out_state ← verbatim;
    break;
  case '/': C_putc('/'); out_state ← post_slash; break;
  case '*':
    if (out_state ≡ post_slash) C_putc('␣');
    /*␣fall␣through␣*/
  default: C_putc(cur_char); out_state ← normal; break;
  }
}
```

56. ⟨ Cases like `!=` 56 ⟩ ≡

```
case plus_plus: C_putc('+'); C_putc('+'); out_state ← normal; break;
case minus_minus: C_putc('-'); C_putc('-'); out_state ← normal; break;
case minus_gt: C_putc('-'); C_putc('>'); out_state ← normal; break;
case gt_gt: C_putc('>'); C_putc('>'); out_state ← normal; break;
case eq_eq: C_putc('='); C_putc('='); out_state ← normal; break;
case lt_lt: C_putc('<'); C_putc('<'); out_state ← normal; break;
case gt_eq: C_putc('>'); C_putc('='); out_state ← normal; break;
case lt_eq: C_putc('<'); C_putc('='); out_state ← normal; break;
case non_eq: C_putc('!'); C_putc('='); out_state ← normal; break;
case and_and: C_putc('&'); C_putc('&'); out_state ← normal; break;
case or_or: C_putc('|'); C_putc('|'); out_state ← normal; break;
case dot_dot_dot: C_putc('.'); C_putc('.'); C_putc('.'); out_state ← normal; break;
case colon_colon: C_putc(':'); C_putc(':'); out_state ← normal; break;
case period_ast: C_putc('.'); C_putc('*'); out_state ← normal; break;
case minus_gt_ast: C_putc('-'); C_putc('>'); C_putc('*'); out_state ← normal; break;
```

This code is used in section 55.

57. When an identifier is output to the C file, characters in the range 128–255 must be changed into something else, so the C compiler won't complain. By default, `CTANGLE` converts the character with code $16x + y$ to the three characters '`X`xy', but a different transliteration table can be specified. Thus a German might want *grün* to appear as a still readable `gruen`. This makes debugging a lot less confusing.

```
#define translit_length 10
```

⟨ Private variables 20 ⟩ +≡

```
static char translit[128][translit_length];
```

58. ⟨ Set initial values 21 ⟩ +≡

```
{
  int i;
  for (i ← 0; i < 128; i++) sprintf(translit[i], "X%02X", (unsigned int)(128 + i));
}
```

59. ⟨ Case of an identifier 59 ⟩ ≡

```
case identifier:
  if (out_state ≡ num_or_id) C_putc('␣');
  for (j ← (cur_val + name_dir)→byte_start, k ← (cur_val + name_dir + 1)→byte_start; j < k; j++)
    if ((eight_bits)(*j) < °200) C_putc(*j);
    else C_printf("%s", translit[(eight_bits)(*j) − °200]);
  out_state ← num_or_id; break;
```

This code is used in section 55.

60. ⟨ Case of a section number 60 ⟩ ≡

```
case section_number:
  if (cur_val > 0) C_printf("/*%d:*/", cur_val);
  else if (cur_val < 0) C_printf("/*:%d*/", -cur_val);
  else if (protect) {
    cur_byte += 4;      ▷ skip line number and file name ◁
    cur_char ← (eight_bits) '\n'; goto restart;
  }
  else {
    sixteen_bits a;
    a ← *cur_byte++ * °400; a += *cur_byte++;      ▷ gets the line number ◁
    C_printf("\n#line␣%d␣\"", (int) a); cur_val ← (int)(*cur_byte++ - °200) * °400;
    cur_val += *cur_byte++;      ▷ points to the file name ◁
    for (j ← (cur_val + name_dir)→byte_start, k ← (cur_val + name_dir + 1)→byte_start; j < k; j++) {
      if (*j ≡ '\\' ∨ *j ≡ '"') C_putc('\\');
      C_putc(*j);
    }
    C_putc('"'); C_putc('\n');
  }
  break;
```

This code is used in section 55.

61. Introduction to the input phase. We have now seen that CTANGLE will be able to output the full C program, if we can only get that program into the byte memory in the proper format. The input process is something like the output process in reverse, since we compress the text as we read it in and we expand it as we write it out.

There are three main input routines. The most interesting is the one that gets the next token of a C text; the other two are used to scan rapidly past TEX text in the CWEB source code. One of the latter routines will jump to the next token that starts with '@', and the other skips to the end of a C comment.

62. Control codes in CWEB begin with '@', and the next character identifies the code. Some of these are of interest only to CWEAVE, so CTANGLE ignores them; the others are converted by CTANGLE into internal code numbers by the *ccode* table below. The ordering of these internal code numbers has been chosen to simplify the program logic; larger numbers are given to the control codes that denote more significant milestones.

```
#define ignore °0 ▷ control code of no interest to CTANGLE ◁
#define ord °302 ▷ control code for '@'' ◁
#define control_text °303 ▷ control code for '@t', '@^', etc. ◁
#define translit_code °304 ▷ control code for '@l' ◁
#define output_defs_code °305 ▷ control code for '@h' ◁
#define format_code °306 ▷ control code for '@f' ◁
#define definition °307 ▷ control code for '@d' ◁
#define begin_C °310 ▷ control code for '@c' ◁
#define section_name °311 ▷ control code for '@<' ◁
#define new_section °312 ▷ control code for '@␣' and '@*' ◁
⟨ Private variables 20 ⟩ +≡
  static eight_bits ccode[256]; ▷ meaning of a char following @ ◁
```

63. ⟨ Set initial values 21 ⟩ +≡

```
{
  int c; ▷ must be int so the for loop will end ◁
  for (c ← 0; c < 256; c++) ccode[c] ← ignore;
}
ccode['␣'] ← ccode['\t'] ← ccode['\n'] ← ccode['\v'] ← ccode['\r'] ← ccode['\f'] ← ccode['*'] ←
    new_section; ccode['@'] ← (eight_bits) '@'; ccode['='] ← string;
ccode['d'] ← ccode['D'] ← definition;
ccode['f'] ← ccode['F'] ← ccode['s'] ← ccode['S'] ← format_code;
ccode['c'] ← ccode['C'] ← ccode['p'] ← ccode['P'] ← begin_C;
ccode['^'] ← ccode[':'] ← ccode['.'] ← ccode['t'] ← ccode['T'] ← ccode['q'] ← ccode['Q'] ←
    control_text; ccode['h'] ← ccode['H'] ← output_defs_code;
ccode['l'] ← ccode['L'] ← translit_code; ccode['&'] ← join; ccode['<'] ← ccode['('] ← section_name;
ccode['\''] ← ord;
```

64. The *skip_ahead* procedure reads through the input at fairly high speed until finding the next non-ignorable control code, which it returns.

```
static eight_bits skip_ahead(void)      ▷ skip to next control code ◁
{
  eight_bits c;      ▷ control code found ◁
  while (true) {
    if (loc > limit ∧ (get_line( ) ≡ false)) return new_section;
    *(limit + 1) ← '@';
    while (*loc ≠ '@') loc++;
    if (loc ≤ limit) {
      loc++; c ← ccode[(eight_bits) *loc]; loc++;
      if (c ≠ ignore ∨ *(loc − 1) ≡ '>') return c;
    }
  }
}
```

65. ⟨ Predeclaration of procedures 8 ⟩ +≡

```
static eight_bits skip_ahead(void);
static boolean skip_comment(boolean);
```

66. The *skip_comment* procedure reads through the input at somewhat high speed in order to pass over comments, which `CTANGLE` does not transmit to the output. If the comment is introduced by `/*`, *skip_comment* proceeds until finding the end-comment token `*/` or a newline; in the latter case *skip_comment* will be called again by *get_next*, since the comment is not finished. This is done so that each newline in the C part of a section is copied to the output; otherwise the #**line** commands inserted into the C file by the output routines become useless. On the other hand, if the comment is introduced by `//` (i.e., if it is a C++ "short comment"), it always is simply delimited by the next newline. The boolean argument *is_long_comment* distinguishes between the two types of comments.

If *skip_comment* comes to the end of the section, it prints an error message. No comment, long or short, is allowed to contain '`@␣`' or '`@*`'.

⟨ Private variables 20 ⟩ +≡

```
static boolean comment_continues ← false;      ▷ are we scanning a comment? ◁
```

```
67.  static boolean skip_comment(      ▷ skips over comments ◁
       boolean 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++;
      }
    }
  }
```

68. Inputting the next token.

```
⟨ Private variables 20 ⟩ +≡
  static name_pointer cur_section_name;      ▷ name of section just scanned ◁
  static boolean no_where;      ▷ suppress print_where? ◁
```

69. As one might expect, *get_next* consists mostly of a big switch that branches to the various special cases that can arise.

```
  static eight_bits get_next(void)      ▷ produces the next input token ◁
  {
    static boolean preprocessing ← false;
    eight_bits c;      ▷ the current character ◁
    while (true) {
      if (loc > limit) {
        if (preprocessing ∧ *(limit − 1) ≠ '\\') preprocessing ← false;
        if (get_line() ≡ false) return new_section;
        else if (print_where ∧ ¬no_where) {
          print_where ← false; ⟨ Insert the line number into tok_mem 85 ⟩
        }
        else return (eight_bits) '\n';
      }
      c ← (eight_bits) *loc;
      if (comment_continues ∨ (c ≡ '/' ∧ (*(loc + 1) ≡ '*' ∨ *(loc + 1) ≡ '/'))) {
        if (skip_comment(comment_continues ∨ *(loc + 1) ≡ '*')) return '\n';
            ▷ scan to end of comment or newline ◁
        else continue;
      }
      loc++;
      if (xisdigit(c) ∨ c ≡ '.') ⟨ Get a constant 73 ⟩
      else if (c ≡ '\'' ∨ c ≡ '"' ∨ ((c ≡ 'L' ∨ c ≡ 'u' ∨ c ≡ 'U') ∧ (*loc ≡ '\'' ∨ *loc ≡ '"'))
          ∨ ((c ≡ 'u' ∧ *loc ≡ '8') ∧ (*(loc + 1) ≡ '\'' ∨ *(loc + 1) ≡ '"'))) ⟨ Get a string 74 ⟩
      else if (isalpha((int) c) ∨ isxalpha(c) ∨ ishigh(c)) ⟨ Get an identifier 72 ⟩
      else if (c ≡ '@') ⟨ Get control code and possible section name 75 ⟩
      else if (xisspace(c)) {
        if (¬preprocessing ∨ loc > limit) continue;      ▷ we don't want a blank after a final backslash ◁
        else return (eight_bits) '␣';      ▷ ignore spaces and tabs, unless preprocessing ◁
      }
      else if (c ≡ '#' ∧ loc ≡ buffer + 1) preprocessing ← true;
    mistake: ⟨ Compress two-symbol operator 71 ⟩
      return c;
    }
  }
```

70. ⟨ Predeclaration of procedures 8 ⟩ +≡ **static eight_bits** *get_next*(**void**);

71. The following code assigns values to the combinations `++`, `--`, `->`, `>=`, `<=`, `==`, `<<`, `>>`, `!=`, `||` and `&&`, and to the C++ combinations `...`, `::`, `.*` and `->*`. The compound assignment operators (e.g., `+=`) are treated as separate tokens.

```
⟨ Compress two-symbol operator 71 ⟩ ≡
  switch (c) {
  case '+':
    if (*loc ≡ '+') compress(plus_plus);
    break;
  case '-':
    if (*loc ≡ '-') { compress(minus_minus); }
    else if (*loc ≡ '>') {
      if (*(loc + 1) ≡ '*') {
        loc++; compress(minus_gt_ast);
      }
      else compress(minus_gt);
    }
    break;
  case '.':
    if (*loc ≡ '*') { compress(period_ast); }
    else if (*loc ≡ '.' ∧ *(loc + 1) ≡ '.') {
      loc++; compress(dot_dot_dot);
    }
    break;
  case ':':
    if (*loc ≡ ':') compress(colon_colon);
    break;
  case '=':
    if (*loc ≡ '=') compress(eq_eq);
    break;
  case '>':
    if (*loc ≡ '=') { compress(gt_eq); }
    else if (*loc ≡ '>') compress(gt_gt);
    break;
  case '<':
    if (*loc ≡ '=') { compress(lt_eq); }
    else if (*loc ≡ '<') compress(lt_lt);
    break;
  case '&':
    if (*loc ≡ '&') compress(and_and);
    break;
  case '|':
    if (*loc ≡ '|') compress(or_or);
    break;
  case '!':
    if (*loc ≡ '=') compress(non_eq);
    break;
  }
```

This code is used in section 69.

72. ⟨ Get an identifier 72 ⟩ ≡

```
{
  id_first ← −−loc; do ++loc; while (isalpha((int) *loc) ∨ isdigit((int) *loc)
      ∨ isxalpha(*loc) ∨ ishigh(*loc)); id_loc ← loc; return identifier;
}
```

This code is used in section 69.

73. ⟨ Get a constant 73 ⟩ ≡

```
{
  boolean hex_flag ← false;      ▷ are we reading a hexadecimal literal? ◁
  id_first ← loc − 1;
  if (*id_first ≡ '.' ∧ ¬xisdigit(*loc)) goto mistake;      ▷ not a constant ◁
  if (*id_first ≡ '0') {
    if (*loc ≡ 'x' ∨ *loc ≡ 'X') {      ▷ hex constant ◁
      hex_flag ← true; loc++;
      while (xisxdigit(*loc) ∨ *loc ≡ '\'') loc++;
    }
    else if (*loc ≡ 'b' ∨ *loc ≡ 'B') {      ▷ binary constant ◁
      loc++;
      while (*loc ≡ '0' ∨ *loc ≡ '1' ∨ *loc ≡ '\'') loc++;
      goto found;
    }
  }
  while (xisdigit(*loc) ∨ *loc ≡ '\'') loc++;
  if (*loc ≡ '.') {
    loc++;
    while ((hex_flag ∧ xisxdigit(*loc)) ∨ xisdigit(*loc) ∨ *loc ≡ '\'') loc++;
  }
  if (*loc ≡ 'e' ∨ *loc ≡ 'E') {      ▷ float constant ◁
    if (*++loc ≡ '+' ∨ *loc ≡ '-') loc++;
    while (xisdigit(*loc) ∨ *loc ≡ '\'') loc++;
  }
  else if (hex_flag ∧ (*loc ≡ 'p' ∨ *loc ≡ 'P')) {      ▷ hex float constant ◁
    if (*++loc ≡ '+' ∨ *loc ≡ '-') loc++;
    while (xisxdigit(*loc) ∨ *loc ≡ '\'') loc++;
  }
found:
  while (*loc ≡ 'u' ∨ *loc ≡ 'U' ∨ *loc ≡ 'l' ∨ *loc ≡ 'L' ∨ *loc ≡ 'f' ∨ *loc ≡ 'F') loc++;
  id_loc ← loc; return constant;
}
```

This code is used in section 69.

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) {
      fputs("\n! String too long: ", stdout); 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.

77.

```
⟨ Scan the section name and make cur_section_name point to it 77 ⟩ ≡
  {
    char *k ← section_text;      ▷ pointer into section_text ◁
    ⟨ Put section name into section_text 79 ⟩
    if (k − section_text > 3 ∧ strncmp(k − 2, "...", 3) ≡ 0)
      cur_section_name ← section_lookup(section_text + 1, k − 3, true);      ▷ true means it's a prefix ◁
    else cur_section_name ← section_lookup(section_text + 1, k, false);      ▷ false means it's not ◁
    if (cur_section_name_char ≡ '(')
      ⟨ If it's not there, add cur_section_name to the output file stack, or complain we're out of room 47 ⟩
    return section_name;
  }
```

This code is used in section 75.

78. Section names are placed into the *section_text* array with consecutive spaces, tabs, and carriage-returns replaced by single spaces. There will be no spaces at the beginning or the end. (We set *section_text*[0] ← '␣' to facilitate this, since the *section_lookup* routine uses *section_text*[1] as the first character of the name.)

```
⟨ Set initial values 21 ⟩ +≡
  section_text[0] ← '␣';
```

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) {
    fputs("\n!␣Section␣name␣too␣long:␣", stdout); 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)
{
  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) app_repl('␣');
      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;
}
```

84. ⟨ Predeclaration of procedures 8 ⟩ +≡ **static void** *scan_repl*(**eight_bits**);

85. Here is the code for the line number: first a **sixteen_bits** equal to °*150000*; then the numeric line number; then a pointer to the file name.

```
#define store_id(a) a ← id_lookup(id_first, id_loc, '\0') − name_dir;
        app_repl((a/°400) + °200); app_repl(a % °400)
⟨ Insert the line number into tok_mem 85 ⟩ ≡
  {
    eight_bits a;      ▷ shadow variable a ◁
    store_two_bytes(°150000);
    if (changing ∧ include_depth ≡ change_depth) {      ▷ correction made Feb 2017 ◁
      id_first ← change_file_name; store_two_bytes((sixteen_bits) change_line);
    } else {
      id_first ← cur_file_name; store_two_bytes((sixteen_bits) cur_line);
    }
    id_loc ← id_first + strlen(id_first); store_id(a);
  }
```

This code is used in sections 69, 83, and 86.

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.

90. Scanning a section. The *scan_section* procedure starts when '`@␣`' or '`@*`' has been sensed in the input, and it proceeds until the end of that section. It uses *section_count* to keep track of the current section number; with luck, `CWEAVE` and `CTANGLE` will both assign the same numbers to sections.

The body of *scan_section* is a loop where we look for control codes that are significant to `CTANGLE`: those that delimit a definition, the C part of a module, or a new module.

```
static void scan_section(void)
{
  name_pointer p;      ▷ section name for the current section ◁
  text_pointer q;      ▷ text for the current section ◁
  sixteen_bits a;      ▷ token for left-hand side of definition ◁
  section_count++; no_where ← true;
  if (*(loc − 1) ≡ '*' ∧ show_progress) {      ▷ starred section ◁
    printf("*%d", (int) section_count); update_terminal;
  }
  next_control ← ignore;
  while (true) {
    ⟨ Skip ahead until next_control corresponds to @d, @<, @␣ or the like 92 ⟩
    if (next_control ≡ definition) {      ▷ @d ◁
      ⟨ Scan a definition 93 ⟩
      continue;
    }
    if (next_control ≡ begin_C) {      ▷ @c or @p ◁
      p ← name_dir; break;
    }
    if (next_control ≡ section_name) {      ▷ @< or @( ◁
      p ← cur_section_name; ⟨ If section is not being defined, continue 94 ⟩
      break;
    }
    return;      ▷ @␣ or @* ◁
  }
  no_where ← print_where ← false; ⟨ Scan the C part of the current section 95 ⟩
}
```

91. ⟨ Predeclaration of procedures 8 ⟩ +≡ `static void scan_section(void);`

92. At the top of this loop, if *next_control* ≡ *section_name*, the section name has already been scanned (see ⟨ Get control code and possible section name 75 ⟩). Thus, if we encounter *next_control* ≡ *section_name* in the skip-ahead process, we should likewise scan the section name, so later processing will be the same in both cases.

```
⟨ Skip ahead until next_control corresponds to @d, @<, @␣ or the like 92 ⟩ ≡
  while (next_control < definition)      ▷ definition is the lowest of the "significant" codes ◁
    if ((next_control ← skip_ahead()) ≡ section_name) {
      loc −= 2; next_control ← get_next();
    }
```

This code is used in section 90.

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.

94. If the section name is not followed by `=` or `+=`, no C code is forthcoming: the section is being cited, not being defined. This use is illegal after the definition part of the current section has started, except inside a comment, but `CTANGLE` does not enforce this rule; it simply ignores the offending section name and everything following it, up to the next significant control code.

⟨ If section is not being defined, **continue** 94 ⟩ ≡
```
while ((next_control ← get_next()) ≡ '+') ;      ▷ allow optional += ◁
if (next_control ≠ '=' ∧ next_control ≠ eq_eq) continue;
```
This code is used in section 90.

95. ⟨ Scan the C part of the current section 95 ⟩ ≡
```
⟨ Insert the section number into tok_mem 96 ⟩
scan_repl(section_name);      ▷ now cur_text points to the replacement text ◁
⟨ Update the data structure so that the replacement text is accessible 97 ⟩
```
This code is used in section 90.

96. ⟨ Insert the section number into *tok_mem* 96 ⟩ ≡
```
store_two_bytes((sixteen_bits)(°150000 + section_count));      ▷ °150000 ≡ °320 ∗ °400 ◁
```
This code is used in section 95.

97. ⟨ Update the data structure so that the replacement text is accessible 97 ⟩ ≡
```
if (p ≡ name_dir ∨ p ≡ Λ) {      ▷ unnamed section, or bad section name ◁
  last_unnamed→text_link ← cur_text − text_info; last_unnamed ← cur_text;
}
else if (p→equiv ≡ (void ∗) text_info) p→equiv ← (void ∗) cur_text;      ▷ first section of this name ◁
else {
  q ← (text_pointer) p→equiv;
  while (q→text_link < section_flag) q ← q→text_link + text_info;      ▷ find end of list ◁
  q→text_link ← cur_text − text_info;
}
cur_text→text_link ← section_flag;      ▷ mark this replacement text as a nonmacro ◁
```
This code is used in section 95.

98.
```
static void phase_one(void)
{
  phase ← 1; section_count ← 0; reset_input(); skip_limbo();
  while (¬input_has_ended) scan_section();
  check_complete();
}
```

99. ⟨ Predeclaration of procedures 8 ⟩ +≡
```
static void phase_one(void);
```

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");
      }
    }
  }
}
```

101. ⟨ Predeclaration of procedures 8 ⟩ +≡ `static void skip_limbo(void);`

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) ∨ *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. 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.