1. Introduction. This file contains code common to both CTANGLE and CWEAVE, which roughly concerns the following problems: character uniformity, input routines, error handling and parsing of command line. We have tried to concentrate in this file all the system dependencies, so as to maximize portability.

In the texts below we will sometimes use CWEB to refer to either of the two component programs, if no confusion can arise.

The file begins with a few basic definitions.

```
⟨ Include files 5 ⟩
⟨ Preprocessor definitions ⟩
⟨ Definitions that should agree with CTANGLE and CWEAVE 2 ⟩
⟨ Other definitions 3 ⟩
⟨ Predeclaration of procedures 33 ⟩
```

2. In certain cases CTANGLE and CWEAVE should do almost, but not quite, the same thing. In these cases we've written common code for both, differentiating between the two by means of the global variable *program*.

```
#define ctangle 0
#define cweave 1
⟨ Definitions that should agree with CTANGLE and CWEAVE 2 ⟩ ≡
  typedef short boolean;
  boolean program;     /* CWEAVE or CTANGLE? */
```

See also sections 7, 10, 20, 27, 29, 32, 56, 67, and 77.

This code is used in section 1.

3. CWEAVE operates in three phases: First it inputs the source file and stores cross-reference data, then it inputs the source once again and produces the TEX output file, and finally it sorts and outputs the index. Similarly, CTANGLE operates in two phases. The global variable *phase* tells which phase we are in.

```
⟨ Other definitions 3 ⟩ ≡
  int phase;     /* which phase are we in? */
```

See also section 11.

This code is used in section 1.

4. There's an initialization procedure that gets both CTANGLE and CWEAVE off to a good start. We will fill in the details of this procedure later.

```
void common_init( )
{
  ⟨ Initialize pointers 30 ⟩;
  ⟨ Set the default options common to CTANGLE and CWEAVE 68 ⟩;
  ⟨ Scan arguments and open output files 78 ⟩;
}
```

5. The character set. CWEB uses the conventions of C programs found in the standard `ctype.h` header file.

⟨ Include files 5 ⟩ ≡
```
#include <ctype.h>
```
See also sections 8 and 22.

This code is used in section 1.

6. A few character pairs are encoded internally as single characters, using the definitions below. These definitions are consistent with an extension of ASCII code originally developed at MIT and explained in Appendix C of *The TEXbook*; thus, users who have such a character set can type things like ≠ and ∧ instead of `!=` and `&&`. (However, their files will not be too portable until more people adopt the extended code.)

If the character set is not ASCII, the definitions given here may conflict with existing characters; in such cases, other arbitrary codes should be substituted. The indexes to CTANGLE and CWEAVE mention every case where similar codes may have to be changed in order to avoid character conflicts. Look for the entry "ASCII code dependencies" in those indexes.

```
#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 not_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 ∞ */
#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 ↰ */
```

7. Input routines. The lowest level of input to the CWEB programs is performed by *input_ln*, which must be told which file to read from. The return value of *input_ln* is 1 if the read is successful and 0 if not (generally this means the file has ended). The conventions of TEX are followed; i.e., the characters of the next line of the file are copied into the *buffer* array, and the global variable *limit* is set to the first unoccupied position. Trailing blanks are ignored. The value of *limit* must be strictly less than *buf_size*, so that *buffer*[*buf_size* − 1] is never filled.

Since *buf_size* is strictly less than *long_buf_size*, some of CWEB's routines use the fact that it is safe to refer to ∗(*limit* + 2) without overstepping the bounds of the array.

```
#define buf_size 100 /* for CWEAVE and CTANGLE */
#define longest_name 1000
#define long_buf_size (buf_size + longest_name) /* for CWEAVE */
#define xisspace(c) (isspace(c) ∧ ((unsigned char) c < °200))
#define xisupper(c) (isupper(c) ∧ ((unsigned char) c < °200))
⟨ Definitions that should agree with CTANGLE and CWEAVE 2 ⟩ +≡
  char buffer[long_buf_size]; /* where each line of input goes */
  char *buffer_end ← buffer + buf_size − 2; /* end of buffer */
  char *limit ← buffer; /* points to the last character in the buffer */
  char *loc ← buffer; /* points to the next character to be read from the buffer */
```

8. ⟨ Include files 5 ⟩ +≡

```
#include <stdio.h>
```

9. In the unlikely event that your standard I/O library does not support *feof*, *getc*, and *ungetc* you may have to change things here.

```
int input_ln(fp) /* copies a line into buffer or returns 0 */
    FILE *fp; /* what file to read from */
{
  register int c ← EOF; /* character read; initialized so some compilers won't complain */
  register char *k; /* where next character goes */
  if (feof(fp)) return (0); /* we have hit end-of-file */
  limit ← k ← buffer; /* beginning of buffer */
  while (k ≤ buffer_end ∧ (c ← getc(fp)) ≠ EOF ∧ c ≠ '\n')
    if ((*(k++) ← c) ≠ '␣') limit ← k;
  if (k > buffer_end)
    if ((c ← getc(fp)) ≠ EOF ∧ c ≠ '\n') {
      ungetc(c, fp);
      loc ← buffer;
      err_print("!␣Input␣line␣too␣long");
    }
  if (c ≡ EOF ∧ limit ≡ buffer) return (0); /* there was nothing after the last newline */
  return (1);
}
```

10. Now comes the problem of deciding which file to read from next. Recall that the actual text that `CWEB` should process comes from two streams: a *web_file*, which can contain possibly nested include commands `@i`, and a *change_file*, which might also contain includes. The *web_file* together with the currently open include files form a stack *file*, whose names are stored in a parallel stack *file_name*. The boolean *changing* tells whether or not we're reading from the *change_file*.

The line number of each open file is also kept for error reporting and for the benefit of `CTANGLE`.

```
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 */
⟨ Definitions that should agree with CTANGLE and CWEAVE 2 ⟩ +≡
  int include_depth; /* current level of nesting */
  FILE *file[max_include_depth]; /* stack of non-change files */
  FILE *change_file; /* change file */
  char file_name[max_include_depth][max_file_name_length]; /* stack of non-change file names */
  char change_file_name[max_file_name_length]; /* name of change file */
  char alt_web_file_name[max_file_name_length]; /* alternate name to try */
  int line[max_include_depth]; /* number of current line in the stacked files */
  int change_line; /* number of current line in change file */
  int change_depth; /* where @y originated during a change */
  boolean input_has_ended; /* if there is no more input */
  boolean changing; /* if the current line is from change_file */
  boolean web_file_open ← 0; /* if the web file is being read */
```

11. When *changing* ≡ 0, the next line of *change_file* is kept in *change_buffer*, for purposes of comparison with the next line of *cur_file*. After the change file has been completely input, we set *change_limit* ← *change_buffer*, so that no further matches will be made.

Here's a shorthand expression for inequality between the two lines:

```
#define lines_dont_match
    (change_limit − change_buffer ≠ limit − buffer ∨ strncmp(buffer, change_buffer, limit − buffer))
⟨ Other definitions 3 ⟩ +≡
  char change_buffer[buf_size]; /* next line of change_file */
  char *change_limit; /* points to the last character in change_buffer */
```

12. Procedure *prime_the_change_buffer* sets *change_buffer* in preparation for the next matching operation. Since blank lines in the change file are not used for matching, we have (*change_limit* ≡ *change_buffer* ∧ ¬*changing*) if and only if the change file is exhausted. This procedure is called only when *changing* is 1; hence error messages will be reported correctly.

```
void prime_the_change_buffer( )
{
  change_limit ← change_buffer; /* this value is used if the change file ends */
  ⟨ Skip over comment lines in the change file; return if end of file 13 ⟩;
  ⟨ Skip to the next nonblank line; return if end of file 14 ⟩;
  ⟨ Move buffer and limit to change_buffer and change_limit 15 ⟩;
}
```

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

```
⟨ Skip over comment lines in the change file; return if end of file 13 ⟩ ≡
  while (1) {
    change_line++;
    if (¬input_ln(change_file)) return;
    if (limit < buffer + 2) continue;
    if (buffer[0] ≠ '@') continue;
    if (xisupper(buffer[1])) buffer[1] ← tolower(buffer[1]);
    if (buffer[1] ≡ 'x') break;
    if (buffer[1] ≡ 'y' ∨ buffer[1] ≡ 'z' ∨ buffer[1] ≡ 'i') {
      loc ← buffer + 2;
      err_print("! Missing @x in change file");
    }
  }
```

This code is used in section 12.

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

```
⟨ Skip to the next nonblank line; return if end of file 14 ⟩ ≡
  do {
    change_line++;
    if (¬input_ln(change_file)) {
      err_print("! Change file ended after @x");
      return;
    }
  } while (limit ≡ buffer);
```

This code is used in section 12.

15. ⟨ Move *buffer* and *limit* to *change_buffer* and *change_limit* 15 ⟩ ≡

```
{
  change_limit ← change_buffer + (limit − buffer);
  strncpy(change_buffer, buffer, limit − buffer + 1);
}
```

This code is used in sections 12 and 16.

16. The following procedure is used to see if the next change entry should go into effect; it is called only when *changing* is 0. The idea is to test whether or not the current contents of *buffer* matches the current contents of *change_buffer*. If not, there's nothing more to do; but if so, a change is called for: All of the text down to the `@y` is supposed to match. An error message is issued if any discrepancy is found. Then the procedure prepares to read the next line from *change_file*.

When a match is found, the current section is marked as changed unless the first line after the `@x` and after the `@y` both start with either `'@*'` or `'@␣'` (possibly preceded by whitespace).

This procedure is called only when *buffer* < *limit*, i.e., when the current line is nonempty.

```
#define if_section_start_make_pending(b)
        { *limit ← '!';
          for (loc ← buffer; xisspace(*loc); loc++) ;
          *limit ← '␣';
          if (*loc ≡ '@' ∧ (xisspace(*(loc + 1)) ∨ *(loc + 1) ≡ '*')) change_pending ← b;
        }
void check_change( )      /* switches to change_file if the buffers match */
{
  int n ← 0;      /* the number of discrepancies found */
  if (lines_dont_match) return;
  change_pending ← 0;
  if (¬changed_section[section_count]) {
    if_section_start_make_pending(1);
    if (¬change_pending) changed_section[section_count] ← 1;
  }
  while (1) {
    changing ← 1;
    print_where ← 1;
    change_line++;
    if (¬input_ln(change_file)) {
      err_print("!␣Change␣file␣ended␣before␣@y");
      change_limit ← change_buffer;
      changing ← 0;
      return;
    }
    if (limit > buffer + 1 ∧ buffer[0] ≡ '@') {
      char xyz_code ← xisupper(buffer[1]) ? tolower(buffer[1]) : buffer[1];
      ⟨ If the current line starts with @y, report any discrepancies and return 17 ⟩;
    }
    ⟨ Move buffer and limit to change_buffer and change_limit 15 ⟩;
    changing ← 0;
    cur_line++;
    while (¬input_ln(cur_file)) {      /* pop the stack or quit */
      if (include_depth ≡ 0) {
        err_print("!␣CWEB␣file␣ended␣during␣a␣change");
        input_has_ended ← 1;
        return;
      }
      include_depth--;
      cur_line++;
    }
    if (lines_dont_match) n++;
  }
```

```
}
```

17. ⟨ If the current line starts with `@y`, report any discrepancies and **return** 17 ⟩ ≡

```
if (xyz_code ≡ 'x' ∨ xyz_code ≡ 'z') {
  loc ← buffer + 2;
  err_print("! Where is the matching @y?");
}
else if (xyz_code ≡ 'y') {
  if (n > 0) {
    loc ← buffer + 2;
    printf("\n! Hmm... %d ", n);
    err_print("of the preceding lines failed to match");
  }
  change_depth ← include_depth;
  return;
}
```

This code is used in section 16.

18. The *reset_input* procedure, which gets `CWEB` ready to read the user's `CWEB` input, is used at the beginning of phase one of `CTANGLE`, phases one and two of `CWEAVE`.

```
void reset_input( )
{
  limit ← buffer;
  loc ← buffer + 1;
  buffer[0] ← ' ';
  ⟨ Open input files 19 ⟩;
  include_depth ← 0;
  cur_line ← 0;
  change_line ← 0;
  change_depth ← include_depth;
  changing ← 1;
  prime_the_change_buffer( );
  changing ← ¬changing;
  limit ← buffer;
  loc ← buffer + 1;
  buffer[0] ← ' ';
  input_has_ended ← 0;
}
```

19. The following code opens the input files.

⟨ Open input files 19 ⟩ ≡

```
if ((web_file ← fopen(web_file_name, "r")) ≡ Λ) {
  strcpy(web_file_name, alt_web_file_name);
  if ((web_file ← fopen(web_file_name, "r")) ≡ Λ)
    fatal("! Cannot open input file ", web_file_name);
}
web_file_open ← 1;
if ((change_file ← fopen(change_file_name, "r")) ≡ Λ)
  fatal("! Cannot open change file ", change_file_name);
```

This code is used in section 18.

20. The *get_line* procedure is called when *loc* > *limit*; it puts the next line of merged input into the buffer and updates the other variables appropriately. A space is placed at the right end of the line. This procedure returns ¬*input_has_ended* because we often want to check the value of that variable after calling the procedure.

If we've just changed from the *cur_file* to the *change_file*, or if the *cur_file* has changed, we tell `CTANGLE` to print this information in the C file by means of the *print_where* flag.

```
#define max_sections 2000
        /* number of identifiers, strings, section names; must be less than 10240 */
⟨ Definitions that should agree with CTANGLE and CWEAVE 2 ⟩ +≡
  typedef unsigned short sixteen_bits;
  sixteen_bits section_count;      /* the current section number */
  boolean changed_section[max_sections];      /* is the section changed? */
  boolean change_pending;
    /* if the current change is not yet recorded in changed_section[section_count] */
  boolean print_where ← 0;      /* should CTANGLE print line and file info? */
```

21.

```
int get_line( )      /* inputs the next line */
{
restart:
  if (changing ∧ include_depth ≡ change_depth)
    ⟨ Read from change_file and maybe turn off changing 25 ⟩;
  if (¬changing ∨ include_depth > change_depth) {
    ⟨ Read from cur_file and maybe turn on changing 24 ⟩;
    if (changing ∧ include_depth ≡ change_depth) goto restart;
  }
  if (input_has_ended) return 0;
  loc ← buffer;
  *limit ← ' ';
  if (buffer[0] ≡ '@' ∧ (buffer[1] ≡ 'i' ∨ buffer[1] ≡ 'I')) {
    loc ← buffer + 2;
    *limit ← '"';
    while (*loc ≡ ' ' ∨ *loc ≡ '\t') loc++;
    if (loc ≥ limit) {
      err_print("! Include file name not given");
      goto restart;
    }
    if (include_depth ≥ max_include_depth − 1) {
      err_print("! Too many nested includes");
      goto restart;
    }
    include_depth++;      /* push input stack */
    ⟨ Try to open include file, abort push if unsuccessful, go to restart 23 ⟩;
  }
  return 1;
}
```

22. When an `@i` line is found in the *cur_file*, we must temporarily stop reading it and start reading from the named include file. The `@i` line should give a complete file name with or without double quotes. If the environment variable `CWEBINPUTS` is set, or if the compiler flag of the same name was defined at compile time, `CWEB` will look for include files in the directory thus named, if it cannot find them in the current directory. (Colon-separated paths are not supported.) The remainder of the `@i` line after the file name is ignored.

```
#define too_long()
        {
          include_depth--;
          err_print("! Include file name too long");
          goto restart;
        }
⟨ Include files 5 ⟩ +≡
#include <stdlib.h>      /* declaration of getenv and exit */
```

23. ⟨ Try to open include file, abort push if unsuccessful, go to *restart* 23 ⟩ ≡

```
{
  char temp_file_name[max_file_name_length];
  char *cur_file_name_end ← cur_file_name + max_file_name_length − 1;
  char *k ← cur_file_name, *kk;
  int l;      /* length of file name */
  if (*loc ≡ '"') {
    loc++;
    while (*loc ≠ '"' ∧ k ≤ cur_file_name_end) *k++ ← *loc++;
    if (loc ≡ limit) k ← cur_file_name_end + 1;      /* unmatched quote is 'too long' */
  }
  else
    while (*loc ≠ ' ' ∧ *loc ≠ '\t' ∧ *loc ≠ '"' ∧ k ≤ cur_file_name_end) *k++ ← *loc++;
  if (k > cur_file_name_end) too_long( );
  *k ← '\0';
  if ((cur_file ← fopen(cur_file_name, "r")) ≠ Λ) {
    cur_line ← 0;
    print_where ← 1;
    goto restart;      /* success */
  }
  kk ← getenv("CWEBINPUTS");
  if (kk ≠ Λ) {
    if ((l ← strlen(kk)) > max_file_name_length − 2) too_long( );
    strcpy(temp_file_name, kk);
  }
  else {
#ifdef CWEBINPUTS
    if ((l ← strlen(CWEBINPUTS)) > max_file_name_length − 2) too_long( );
    strcpy(temp_file_name, CWEBINPUTS);
#else
    l ← 0;
#endif      /* CWEBINPUTS */
  }
  if (l > 0) {
    if (k + l + 2 ≥ cur_file_name_end) too_long( );
    for ( ; k ≥ cur_file_name; k−−) *(k + l + 1) ← *k;
    strcpy(cur_file_name, temp_file_name);
    cur_file_name[l] ← '/';      /* UNIX pathname separator */
    if ((cur_file ← fopen(cur_file_name, "r")) ≠ Λ) {
      cur_line ← 0;
      print_where ← 1;
      goto restart;      /* success */
    }
  }
  include_depth−−;
  err_print("! Cannot open include file");
  goto restart;
}
```

This code is used in section 21.

24. ⟨ Read from *cur_file* and maybe turn on *changing* 24 ⟩ ≡

```
{
  cur_line++;
  while (¬input_ln(cur_file)) {      /* pop the stack or quit */
    print_where ← 1;
    if (include_depth ≡ 0) {
      input_has_ended ← 1;
      break;
    }
    else {
      fclose(cur_file);
      include_depth−−;
      if (changing ∧ include_depth ≡ change_depth) break;
      cur_line++;
    }
  }
  if (¬changing ∧ ¬input_has_ended)
    if (limit − buffer ≡ change_limit − change_buffer)
      if (buffer[0] ≡ change_buffer[0])
        if (change_limit > change_buffer) check_change( );
}
```

This code is used in section 21.

25. ⟨ Read from *change_file* and maybe turn off *changing* 25 ⟩ ≡

```
{
  change_line++;
  if (¬input_ln(change_file)) {
    err_print("! Change file ended without @z");
    buffer[0] ← '@';
    buffer[1] ← 'z';
    limit ← buffer + 2;
  }
  if (limit > buffer) {      /* check if the change has ended */
    if (change_pending) {
      if_section_start_make_pending(0);
      if (change_pending) {
        changed_section[section_count] ← 1;
        change_pending ← 0;
      }
    }
    *limit ← ' ';
    if (buffer[0] ≡ '@') {
      if (xisupper(buffer[1])) buffer[1] ← tolower(buffer[1]);
      if (buffer[1] ≡ 'x' ∨ buffer[1] ≡ 'y') {
        loc ← buffer + 2;
        err_print("! Where is the matching @z?");
      }
      else if (buffer[1] ≡ 'z') {
        prime_the_change_buffer( );
        changing ← ¬changing;
        print_where ← 1;
      }
    }
  }
}
```

This code is used in section 21.

26. At the end of the program, we will tell the user if the change file had a line that didn't match any relevant line in *web_file*.

```
void check_complete( )
{
  if (change_limit ≠ change_buffer) {      /* changing is 0 */
    strncpy(buffer, change_buffer, change_limit − change_buffer + 1);
    limit ← buffer + (int)(change_limit − change_buffer);
    changing ← 1;
    change_depth ← include_depth;
    loc ← buffer;
    err_print("! Change file entry did not match");
  }
}
```

27. Storage of names and strings. Both `CWEAVE` and `CTANGLE` store identifiers, section names and other strings in a large array of **char**s, called *byte_mem*. Information about the names is kept in the array *name_dir*, whose elements are structures of type *name_info*, containing a pointer into the *byte_mem* array (the address where the name begins) and other data. A *name_pointer* variable is a pointer into *name_dir*.

```
#define max_bytes 90000
        /* the number of bytes in identifiers, index entries, and section names; must be less than 2^24 */
#define max_names 4000
        /* number of identifiers, strings, section names; must be less than 10240 */
⟨ Definitions that should agree with CTANGLE and CWEAVE 2 ⟩ +≡
  typedef struct name_info {
    char *byte_start;      /* beginning of the name in byte_mem */
    ⟨ More elements of name_info structure 31 ⟩
  } name_info;      /* contains information about an identifier or section name */
  typedef name_info *name_pointer;      /* pointer into array of name_infos */
  char byte_mem[max_bytes];      /* characters of names */
  char *byte_mem_end ← byte_mem + max_bytes − 1;      /* end of byte_mem */
  name_info name_dir[max_names];      /* information about names */
  name_pointer name_dir_end ← name_dir + max_names − 1;      /* end of name_dir */
```

28. The actual sequence of characters in the name pointed to by a **name_pointer** p appears in positions $p\text{→}byte\_start$ to $(p+1)\text{→}byte\_start - 1$, inclusive. The *print_id* macro prints this text on the user's terminal.

```
#define length(c) (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 */
```

29. The first unused position in *byte_mem* and *name_dir* is kept in *byte_ptr* and *name_ptr*, respectively. Thus we usually have $name\_ptr\text{→}byte\_start \equiv byte\_ptr$, and certainly we want to keep $name\_ptr \le name\_dir\_end$ and $byte\_ptr \le byte\_mem\_end$.

```
⟨ Definitions that should agree with CTANGLE and CWEAVE 2 ⟩ +≡
  name_pointer name_ptr;      /* first unused position in byte_start */
  char *byte_ptr;      /* first unused position in byte_mem */
```

30.
```
⟨ Initialize pointers 30 ⟩ ≡
  name_dir→byte_start ← byte_ptr ← byte_mem;      /* position zero in both arrays */
  name_ptr ← name_dir + 1;      /* name_dir[0] will be used only for error recovery */
  name_ptr→byte_start ← byte_mem;      /* this makes name 0 of length zero */
```

See also sections 34 and 41.

This code is used in section 4.

31. The names of identifiers are found by computing a hash address h and then looking at strings of bytes signified by the **name_pointer**s $hash[h]$, $hash[h]\text{→}link$, $hash[h]\text{→}link\text{→}link$, ..., until either finding the desired name or encountering the null pointer.

```
⟨ More elements of name_info structure 31 ⟩ ≡
  struct name_info *link;
```

See also sections 40 and 55.

This code is used in section 27.

32. The hash table itself consists of *hash_size* entries of type **name_pointer**, and is updated by the *id_lookup* procedure, which finds a given identifier and returns the appropriate **name_pointer**. The matching is done by the function *names_match*, which is slightly different in `CWEAVE` and `CTANGLE`. If there is no match for the identifier, it is inserted into the table.

```
#define hash_size 353 /* should be prime */
⟨ Definitions that should agree with CTANGLE and CWEAVE 2 ⟩ +≡
  typedef name_pointer *hash_pointer;
  name_pointer hash[hash_size]; /* heads of hash lists */
  hash_pointer hash_end ← hash + hash_size − 1; /* end of hash */
  hash_pointer h; /* index into hash-head array */
```

33. ⟨ Predeclaration of procedures 33 ⟩ ≡

```
  extern int names_match( );
```

See also sections 38, 46, 53, 57, 60, 63, 69, and 81.

This code is used in section 1.

34. Initially all the hash lists are empty.

```
⟨ Initialize pointers 30 ⟩ +≡
  for (h ← hash; h ≤ hash_end; *h++ ← Λ) ;
```

35. Here is the main procedure for finding identifiers:

```
  name_pointer id_lookup(first, last, t) /* looks up a string in the identifier table */
      char *first; /* first character of string */
      char *last; /* last character of string plus one */
      char t; /* the ilk; used by CWEAVE only */
  {
    char *i ← first; /* position in buffer */
    int h; /* hash code */
    int l; /* length of the given identifier */
    name_pointer p; /* where the identifier is being sought */
    if (last ≡ Λ)
      for (last ← first; *last ≠ '\0'; last++) ;
    l ← last − first; /* compute the length */
    ⟨ Compute the hash code h 36 ⟩;
    ⟨ Compute the name location p 37 ⟩;
    if (p ≡ name_ptr) ⟨ Enter a new name into the table at position p 39 ⟩;
    return (p);
  }
```

36. A simple hash code is used: If the sequence of character codes is $c_1 c_2 \ldots c_n$, its hash value will be

$$(2^{n-1}c_1 + 2^{n-2}c_2 + \cdots + c_n) \bmod hash\_size.$$

```
⟨ Compute the hash code h 36 ⟩ ≡
  h ← (unsigned char) *i;
  while (++i < last) h ← (h + h + (int)((unsigned char) *i)) % hash_size;
```

This code is used in section 35.

37. If the identifier is new, it will be placed in position $p \leftarrow name\_ptr$, otherwise p will point to its existing location.

```
⟨ Compute the name location p 37 ⟩ ≡
  p ← hash[h];
  while (p ∧ ¬names_match(p, first, l, t)) p ← p→link;
  if (p ≡ Λ) {
    p ← name_ptr;      /* the current identifier is new */
    p→link ← hash[h];
    hash[h] ← p;      /* insert p at beginning of hash list */
  }
```

This code is used in section 35.

38. The information associated with a new identifier must be initialized in a slightly different way in `CWEAVE` than in `CTANGLE`; hence the *init_p* procedure.

```
⟨ Predeclaration of procedures 33 ⟩ +≡
  void init_p( );
```

39.

```
⟨ Enter a new name into the table at position p 39 ⟩ ≡
  {
    if (byte_ptr + l > byte_mem_end) overflow("byte␣memory");
    if (name_ptr ≥ name_dir_end) overflow("name");
    strncpy(byte_ptr, first, l);
    (++name_ptr)→byte_start ← byte_ptr += l;
    if (program ≡ cweave) init_p(p, t);
  }
```

This code is used in section 35.

40. The names of sections are stored in *byte_mem* together with the identifier names, but a hash table is not used for them because `CTANGLE` needs to be able to recognize a section name when given a prefix of that name. A conventional binary search tree is used to retrieve section names, with fields called *llink* and *rlink* (where *llink* takes the place of *link*). The root of this tree is stored in *name_dir→rlink*; this will be the only information in *name_dir*[0].

Since the space used by *rlink* has a different function for identifiers than for section names, we declare it as a **union**.

```
#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 */
⟨ More elements of name_info structure 31 ⟩ +≡
  union {
    struct name_info *Rlink;      /* right link in binary search tree for section names */
    char Ilk;      /* used by identifiers in CWEAVE only */
  } dummy;
```

41.

```
⟨ Initialize pointers 30 ⟩ +≡
  root ← Λ;      /* the binary search tree starts out with nothing in it */
```

42. If p is a **name_pointer** variable, as we have seen, $p\neg byte\_start$ is the beginning of the area where the name corresponding to p is stored. However, if p refers to a section name, the name may need to be stored in chunks, because it may "grow": a prefix of the section name may be encountered before the full name. Furthermore we need to know the length of the shortest prefix of the name that was ever encountered.

We solve this problem by inserting two extra bytes at $p\neg byte\_start$, representing the length of the shortest prefix, when p is a section name. Furthermore, the last byte of the name will be a blank space if p is a prefix. In the latter case, the name pointer $p+1$ will allow us to access additional chunks of the name: The second chunk will begin at the name pointer $(p+1)\neg link$, and if it too is a prefix (ending with blank) its *link* will point to additional chunks in the same way. Null links are represented by *name_dir*.

```
#define first_chunk(p) ((p)¬byte_start + 2)
#define prefix_length(p)
        (int)((unsigned char) *((p)¬byte_start) * 256 + (unsigned char) *((p)¬byte_start + 1))
#define set_prefix_length(p, m) (*((p)¬byte_start) ← (m)/256, *((p)¬byte_start + 1) ← (m) % 256)
  void print_section_name(p)
      name_pointer p;
{
  char *ss, *s ← first_chunk(p);
  name_pointer q ← p + 1;
  while (p ≠ name_dir) {
    ss ← (p + 1)¬byte_start − 1;
    if (*ss ≡ '␣' ∧ ss ≥ s) {
      term_write(s, ss − s);
      p ← q¬link;
      q ← p;
    }
    else {
      term_write(s, ss + 1 − s);
      p ← name_dir;
      q ← Λ;
    }
    s ← p¬byte_start;
  }
  if (q) term_write("...", 3);      /* complete name not yet known */
}
```

43.
```
void sprint_section_name(dest, p)
    char *dest;
    name_pointer p;
{
  char *ss, *s ← first_chunk(p);
  name_pointer q ← p + 1;
  while (p ≠ name_dir) {
    ss ← (p + 1)¬byte_start − 1;
    if (*ss ≡ '␣' ∧ ss ≥ s) {
      p ← q¬link;
      q ← p;
    }
    else {
      ss++;
      p ← name_dir;
    }
    strncpy(dest, s, ss − s), dest += ss − s;
    s ← p¬byte_start;
  }
  *dest ← '\0';
}
```

44.
```
void print_prefix_name(p)
    name_pointer p;
{
  char *s ← first_chunk(p);
  int l ← prefix_length(p);
  term_write(s, l);
  if (s + l < (p + 1)¬byte_start) term_write("...", 3);
}
```

45. When we compare two section names, we'll need a function analogous to *strcmp*. But we do not assume the strings are null-terminated, and we keep an eye open for prefixes and extensions.

```
#define less 0      /* the first name is lexicographically less than the second */
#define equal 1     /* the first name is equal to the second */
#define greater 2   /* the first name is lexicographically greater than the second */
#define prefix 3    /* the first name is a proper prefix of the second */
#define extension 4 /* the first name is a proper extension of the second */
int web_strcmp(j, j_len, k, k_len)     /* fuller comparison than strcmp */
    char *j, *k;     /* beginning of first and second strings */
    int j_len, k_len;     /* length of strings */
{
  char *j1 ← j + j_len, *k1 ← k + k_len;
  while (k < k1 ∧ j < j1 ∧ *j ≡ *k) k++, j++;
  if (k ≡ k1)
    if (j ≡ j1) return equal;
    else return extension;
  else if (j ≡ j1) return prefix;
  else if (*j < *k) return less;
  else return greater;
}
```

46. Adding a section name to the tree is straightforward if we know its parent and whether it's the *rlink* or *llink* of the parent. As a special case, when the name is the first section being added, we set the "parent" to Λ. When a section name is created, it has only one chunk, which however may be just a prefix; the full name will hopefully be unveiled later. Obviously, *prefix_length* starts out as the length of the first chunk, though it may decrease later.

The information associated with a new node must be initialized differently in `CWEAVE` and `CTANGLE`; hence the *init_node* procedure, which is defined differently in `cweave.w` and `ctangle.w`.

```
⟨ Predeclaration of procedures 33 ⟩ +≡
  extern void init_node( );
```

47.
```
name_pointer add_section_name(par, c, first, last, ispref)      /* install a new node in the tree */
    name_pointer par;      /* parent of new node */
    int c;      /* right or left? */
    char *first;      /* first character of section name */
    char *last;      /* last character of section name, plus one */
    int ispref;      /* are we adding a prefix or a full name? */
{
  name_pointer p ← name_ptr;      /* new node */
  char *s ← first_chunk(p);
  int name_len ← last − first + ispref;      /* length of section name */
  if (s + name_len > byte_mem_end) overflow("byte␣memory");
  if (name_ptr + 1 ≥ name_dir_end) overflow("name");
  (++name_ptr)→byte_start ← byte_ptr ← s + name_len;
  if (ispref) {
    *(byte_ptr − 1) ← '␣';
    name_len−−;
    name_ptr→link ← name_dir;
    (++name_ptr)→byte_start ← byte_ptr;
  }
  set_prefix_length(p, name_len);
  strncpy(s, first, name_len);
  p→llink ← Λ;
  p→rlink ← Λ;
  init_node(p);
  return par ≡ Λ ? (root ← p) : c ≡ less ? (par→llink ← p) : (par→rlink ← p);
}
```

48.
```
void extend_section_name(p, first, last, ispref)
    name_pointer p;      /* name to be extended */
    char *first;         /* beginning of extension text */
    char *last;          /* one beyond end of extension text */
    int ispref;          /* are we adding a prefix or a full name? */
{
  char *s;
  name_pointer q ← p + 1;
  int name_len ← last − first + ispref;
  if (name_ptr ≥ name_dir_end) overflow("name");
  while (q→link ≠ name_dir) q ← q→link;
  q→link ← name_ptr;
  s ← name_ptr→byte_start;
  name_ptr→link ← name_dir;
  if (s + name_len > byte_mem_end) overflow("byte␣memory");
  (++name_ptr)→byte_start ← byte_ptr ← s + name_len;
  strncpy(s, first, name_len);
  if (ispref) *(byte_ptr − 1) ← '␣';
}
```

49. The *section_lookup* procedure is supposed to find a section name that matches a new name, installing the new name if it doesn't match an existing one. The new name is the string between *first* and *last*; a "match" means that the new name exactly equals or is a prefix or extension of a name in the tree.

```
name_pointer section_lookup(first, last, ispref)    /* find or install section name in tree */
    char *first, *last;    /* first and last characters of new name */
    int ispref;            /* is the new name a prefix or a full name? */
{
  int c ← 0;               /* comparison between two names; initialized so some compilers won't complain */
  name_pointer p ← root;   /* current node of the search tree */
  name_pointer q ← Λ;      /* another place to look in the tree */
  name_pointer r ← Λ;      /* where a match has been found */
  name_pointer par ← Λ;    /* parent of p, if r is Λ; otherwise parent of r */
  int name_len ← last − first + 1;
  ⟨ Look for matches for new name among shortest prefixes, complaining if more than one is found 50 ⟩;
  ⟨ If no match found, add new name to tree 51 ⟩;
  ⟨ If one match found, check for compatibility and return match 52 ⟩;
}
```

50. A legal new name matches an existing section name if and only if it matches the shortest prefix of that section name. Therefore we can limit our search for matches to shortest prefixes, which eliminates the need for chunk-chasing at this stage.

⟨ Look for matches for new name among shortest prefixes, complaining if more than one is found 50 ⟩ ≡

```
while (p) {      /* compare shortest prefix of p with new name */
  c ← web_strcmp(first, name_len, first_chunk(p), prefix_length(p));
  if (c ≡ less ∨ c ≡ greater) {      /* new name does not match p */
    if (r ≡ Λ)      /* no previous matches have been found */
      par ← p;
    p ← (c ≡ less ? p→llink : p→rlink);
  }
  else {      /* new name matches p */
    if (r ≠ Λ) {      /* and also r: illegal */
      printf("\n!␣Ambiguous␣prefix:␣matches␣<");
      print_prefix_name(p);
      printf(">\n␣and␣<");
      print_prefix_name(r);
      err_print(">");
      return name_dir;      /* the unsection */
    }
    r ← p;      /* remember match */
    p ← p→llink;      /* try another */
    q ← r→rlink;      /* we'll get back here if the new p doesn't match */
  }
  if (p ≡ Λ) p ← q, q ← Λ;      /* q held the other branch of r */
}
```

This code is used in section 49.

51. ⟨ If no match found, add new name to tree 51 ⟩ ≡

```
if (r ≡ Λ)      /* no matches were found */
  return add_section_name(par, c, first, last + 1, ispref);
```

This code is used in section 49.

52. Although error messages are given in anomalous cases, we do return the unique best match when a discrepancy is found, because users often change a title in one place while forgetting to change it elsewhere.

```
⟨ If one match found, check for compatibility and return match 52 ⟩ ≡
  switch (section_name_cmp(&first, name_len, r)) {      /* compare all of r with new name */
  case prefix:
    if (¬ispref) {
      printf("\n!␣New␣name␣is␣a␣prefix␣of␣<");
      print_section_name(r);
      err_print(">");
    }
    else if (name_len < prefix_length(r)) set_prefix_length(r, name_len);      /* fall through */
  case equal: return r;
  case extension:
    if (¬ispref ∨ first ≤ last) extend_section_name(r, first, last + 1, ispref);
    return r;
  case bad_extension: printf("\n!␣New␣name␣extends␣<");
    print_section_name(r);
    err_print(">");
    return r;
  default:      /* no match: illegal */
    printf("\n!␣Section␣name␣incompatible␣with␣<");
    print_prefix_name(r);
    printf(">,\n␣which␣abbreviates␣<");
    print_section_name(r);
    err_print(">");
    return r;
  }
```

This code is used in section 49.

53. The return codes of *section_name_cmp*, which compares a string with the full name of a section, are those of *web_strcmp* plus *bad_extension*, used when the string is an extension of a supposedly already complete section name. This function has a side effect when the comparison string is an extension: It advances the address of the first character of the string by an amount equal to the length of the known part of the section name.

The name `@<foo...@>` should be an acceptable "abbreviation" for `@<foo@>`. If such an abbreviation comes after the complete name, there's no trouble recognizing it. If it comes before the complete name, we simply append a null chunk. This logic requires us to regard `@<foo...@>` as an "extension" of itself.

```
#define bad_extension 5
⟨ Predeclaration of procedures 33 ⟩ +≡
  int section_name_cmp( );
```

54.
```
int section_name_cmp(pfirst, len, r)
    char **pfirst;      /* pointer to beginning of comparison string */
    int len;      /* length of string */
    name_pointer r;      /* section name being compared */
{
  char *first ← *pfirst;      /* beginning of comparison string */
  name_pointer q ← r + 1;      /* access to subsequent chunks */
  char *ss, *s ← first_chunk(r);
  int c;      /* comparison */
  int ispref;      /* is chunk r a prefix? */
  while (1) {
    ss ← (r + 1)→byte_start − 1;
    if (*ss ≡ '␣' ∧ ss ≥ r→byte_start) ispref ← 1, q ← q→link;
    else ispref ← 0, ss++, q ← name_dir;
    switch (c ← web_strcmp(first, len, s, ss − s)) {
    case equal:
      if (q ≡ name_dir)
        if (ispref) {
          *pfirst ← first + (ss − s);
          return extension;      /* null extension */
        }
        else return equal;
      else return (q→byte_start ≡ (q + 1)→byte_start) ? equal : prefix;
    case extension:
      if (¬ispref) return bad_extension;
      first += ss − s;
      if (q ≠ name_dir) {
        len −= ss − s;
        s ← q→byte_start;
        r ← q;
        continue;
      }
      *pfirst ← first;
      return extension;
    default: return c;
    }
  }
}
```

55. The last component of **name_info** is different for `CTANGLE` and `CWEAVE`. In `CTANGLE`, 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*. In `CWEAVE`, on the other hand, if p points to an identifier, p→*xref* is a pointer to its list of cross-references, an element of the array *xmem*. The make-up of *text_info* and *xmem* is discussed in the `CTANGLE` and `CWEAVE` source files, respectively; here we just declare a common field *equiv_or_xref* as a pointer to a **char**.

⟨ More elements of **name_info** structure 31 ⟩ +≡
```
  char *equiv_or_xref;      /* info corresponding to names */
```

56. Reporting errors to the user. A global variable called *history* will contain one of four values at the end of every run: *spotless* means that no unusual messages were printed; *harmless_message* means that a message of possible interest was printed but no serious errors were detected; *error_message* means that at least one error was found; *fatal_message* means that the program terminated abnormally. The value of *history* does not influence the behavior of the program; it is simply computed for the convenience of systems that might want to use such information.

```
#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
```

⟨ Definitions that should agree with `CTANGLE` and `CWEAVE` 2 ⟩ +≡

```
int history ← spotless; /* indicates how bad this run was */
```

57. The command '*err_print*(`"!␣Error␣message"`)' will report a syntax error to the user, by printing the error message at the beginning of a new line and then giving an indication of where the error was spotted in the source file. Note that no period follows the error message, since the error routine will automatically supply a period. A newline is automatically supplied if the string begins with `"!"`.

⟨ Predeclaration of procedures 33 ⟩ +≡

```
void err_print( );
```

58.

```
void err_print(s) /* prints '.' and location of error message */
  char *s;
{
  char *k, *l; /* pointers into buffer */
  printf(*s ≡ '!' ? "\n%s" : "%s", s);
  if (web_file_open) ⟨ Print error location based on input buffer 59 ⟩;
  update_terminal;
  mark_error;
}
```

59. The error locations can be indicated by using the global variables *loc*, *cur_line*, *cur_file_name* and *changing*, which tell respectively the first unlooked-at position in *buffer*, the current line number, the current file, and whether the current line is from *change_file* or *cur_file*. This routine should be modified on systems whose standard text editor has special line-numbering conventions.

```
⟨ Print error location based on input buffer 59 ⟩ ≡
  {
    if (changing ∧ include_depth ≡ change_depth) printf(". (l. %d of change file)\n", change_line);
    else if (include_depth ≡ 0) printf(". (l. %d)\n", cur_line);
    else printf(". (l. %d of include file %s)\n", cur_line, cur_file_name);
    l ← (loc ≥ limit ? limit : loc);
    if (l > buffer) {
      for (k ← buffer; k < l; k++)
        if (∗k ≡ '\t') putchar(' ');
        else putchar(∗k);      /∗ print the characters already read ∗/
      putchar('\n');
      for (k ← buffer; k < l; k++) putchar(' ');      /∗ space out the next line ∗/
    }
    for (k ← l; k < limit; k++) putchar(∗k);      /∗ print the part not yet read ∗/
    if (∗limit ≡ '|') putchar('|');      /∗ end of C text in section names ∗/
    putchar(' ');      /∗ to separate the message from future asterisks ∗/
  }
```

This code is used in section 58.

60. When no recovery from some error has been provided, we have to wrap up and quit as graciously as possible. This is done by calling the function *wrap_up* at the end of the code.

CTANGLE and CWEAVE have their own notions about how to print the job statistics.

```
⟨ Predeclaration of procedures 33 ⟩ +≡
  int wrap_up( );
  extern void print_stats( );
```

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

```
int wrap_up( )
{
  putchar('\n');
  if (show_stats) print_stats( );      /∗ print statistics about memory usage ∗/
  ⟨ Print the job history 62 ⟩;
  if (history > harmless_message) return (1);
  else return (0);
}
```

62. ⟨ Print the job *history* 62 ⟩ ≡

```
switch (history) {
case spotless:
  if (show_happiness) printf("(No␣errors␣were␣found.)\n");
  break;
case harmless_message: printf("(Did␣you␣see␣the␣warning␣message␣above?)\n");
  break;
case error_message: printf("(Pardon␣me,␣but␣I␣think␣I␣spotted␣something␣wrong.)\n");
  break;
case fatal_message: printf("(That␣was␣a␣fatal␣error,␣my␣friend.)\n");
}  /* there are no other cases */
```

This code is used in section 61.

63. When there is no way to recover from an error, the *fatal* subroutine is invoked. This happens most often when *overflow* occurs.

⟨ Predeclaration of procedures 33 ⟩ +≡

```
void fatal( ), overflow( );
```

64. The two parameters to *fatal* are strings that are essentially concatenated to print the final error message.

```
void fatal(s, t)
    char *s, *t;
{
  if (*s) printf(s);
  err_print(t);
  history ← fatal_message;
  exit(wrap_up( ));
}
```

65. An overflow stop occurs if CWEB's tables aren't large enough.

```
void overflow(t)
    char *t;
{
  printf("\n!␣Sorry,␣%s␣capacity␣exceeded", t);
  fatal("", "");
}
```

66. Sometimes the program's behavior is far different from what it should be, and CWEB prints an error message that is really for the CWEB maintenance person, not the user. In such cases the program says *confusion*("indication␣of␣where␣we␣are").

```
#define confusion(s) fatal("!␣This␣can't␣happen:␣", s)
```

67. Command line arguments. The user calls `CWEAVE` and `CTANGLE` with arguments on the command line. These are either file names or flags to be turned off (beginning with `"-"`) or flags to be turned on (beginning with `"+"`). The following globals are for communicating the user's desires to the rest of the program. The various file name variables contain strings with the names of those files. Most of the 128 flags are undefined but available for future extensions.

```
#define show_banner flags['b']   /* should the banner line be printed? */
#define show_progress flags['p']   /* should progress reports be printed? */
#define show_stats flags['s']   /* should statistics be printed at end of run? */
#define show_happiness flags['h']   /* should lack of errors be announced? */
```

⟨ Definitions that should agree with `CTANGLE` and `CWEAVE` 2 ⟩ +≡

```
int argc;   /* copy of ac parameter to main */
char **argv;   /* copy of av parameter to main */
char C_file_name[max_file_name_length];   /* name of C_file */
char tex_file_name[max_file_name_length];   /* name of tex_file */
char idx_file_name[max_file_name_length];   /* name of idx_file */
char scn_file_name[max_file_name_length];   /* name of scn_file */
boolean flags[128];   /* an option for each 7-bit code */
```

68. The *flags* will be initially zero. Some of them are set to 1 before scanning the arguments; if additional flags are 1 by default they should be set before calling *common_init*.

⟨ Set the default options common to `CTANGLE` and `CWEAVE` 68 ⟩ ≡

```
show_banner ← show_happiness ← show_progress ← 1;
```

This code is used in section 4.

69. We now must look at the command line arguments and set the file names accordingly. At least one file name must be present: the `CWEB` file. It may have an extension, or it may omit the extension to get `".w"` or `".web"` added. The TEX output file name is formed by replacing the `CWEB` file name extension by `".tex"`, and the C file name by replacing the extension by `".c"`, after removing the directory name (if any).

If there is a second file name present among the arguments, it is the change file, again either with an extension or without one to get `".ch"`. An omitted change file argument means that `"/dev/null"` should be used, when no changes are desired.

If there's a third file name, it will be the output file.

⟨ Predeclaration of procedures 33 ⟩ +≡

```
void scan_args( );
```

70.
```
void scan_args()
{
  char *dot_pos;      /* position of '.' in the argument */
  char *name_pos;     /* file name beginning, sans directory */
  register char *s;   /* register for scanning strings */
  boolean found_web ← 0, found_change ← 0, found_out ← 0;   /* have these names been seen? */
  boolean flag_change;
  while (--argc > 0) {
    if ((**(++argv) ≡ '-' ∨ **argv ≡ '+') ∧ *(*argv + 1)) ⟨ Handle flag argument 74 ⟩
    else {
      s ← name_pos ← *argv; dot_pos ← Λ;
      while (*s) {
        if (*s ≡ '.') dot_pos ← s++;
        else if (*s ≡ '/') dot_pos ← Λ, name_pos ← ++s;
        else s++;
      }
      if (¬found_web) ⟨ Make web_file_name, tex_file_name, and C_file_name 71 ⟩
      else if (¬found_change) ⟨ Make change_file_name from fname 72 ⟩
      else if (¬found_out) ⟨ Override tex_file_name and C_file_name 73 ⟩
      else ⟨ Print usage error message and quit 75 ⟩;
    }
  }
  if (¬found_web) ⟨ Print usage error message and quit 75 ⟩;
  if (found_change ≤ 0) strcpy(change_file_name, "/dev/null");
}
```

71. We use all of **argv* for the *web_file_name* if there is a `'.'` in it, otherwise we add `".w"`. If this file can't be opened, we prepare an *alt_web_file_name* by adding `"web"` after the dot. The other file names come from adding other things after the dot. We must check that there is enough room in *web_file_name* and the other arrays for the argument.

```
⟨ Make web_file_name, tex_file_name, and C_file_name 71 ⟩ ≡
{
  if (s − *argv > max_file_name_length − 5) ⟨ Complain about argument length 76 ⟩;
  if (dot_pos ≡ Λ) sprintf(web_file_name, "%s.w", *argv);
  else {
    strcpy(web_file_name, *argv);
    *dot_pos ← 0;   /* string now ends where the dot was */
  }
  sprintf(alt_web_file_name, "%s.web", *argv);
  sprintf(tex_file_name, "%s.tex", name_pos);   /* strip off directory name */
  sprintf(idx_file_name, "%s.idx", name_pos);
  sprintf(scn_file_name, "%s.scn", name_pos);
  sprintf(C_file_name, "%s.c", name_pos);
  found_web ← 1;
}
```

This code is used in section 70.

72. ⟨ Make *change_file_name* from *fname* 72 ⟩ ≡

```
{
  if (strcmp(*argv, "-") ≡ 0) found_change ← −1;
  else {
    if (s − *argv > max_file_name_length − 4) ⟨ Complain about argument length 76 ⟩;
    if (dot_pos ≡ Λ) sprintf(change_file_name, "%s.ch", *argv);
    else strcpy(change_file_name, *argv);
    found_change ← 1;
  }
}
```

This code is used in section 70.

73. ⟨ Override *tex_file_name* and *C_file_name* 73 ⟩ ≡

```
{
  if (s − *argv > max_file_name_length − 5) ⟨ Complain about argument length 76 ⟩;
  if (dot_pos ≡ Λ) {
    sprintf(tex_file_name, "%s.tex", *argv);
    sprintf(idx_file_name, "%s.idx", *argv);
    sprintf(scn_file_name, "%s.scn", *argv);
    sprintf(C_file_name, "%s.c", *argv);
  }
  else {
    strcpy(tex_file_name, *argv);
    strcpy(C_file_name, *argv);
    if (flags['x']) {      /* indexes will be generated */
      *dot_pos ← 0;
      sprintf(idx_file_name, "%s.idx", *argv);
      sprintf(scn_file_name, "%s.scn", *argv);
    }
  }
  found_out ← 1;
}
```

This code is used in section 70.

74. ⟨ Handle flag argument 74 ⟩ ≡

```
{
  if (**argv ≡ '-') flag_change ← 0;
  else flag_change ← 1;
  for (dot_pos ← *argv + 1; *dot_pos > '\0'; dot_pos++) flags[*dot_pos] ← flag_change;
}
```

This code is used in section 70.

75. ⟨ Print usage error message and quit 75 ⟩ ≡

```
{
  if (program ≡ ctangle)
    fatal("!␣Usage:␣ctangle␣[options]␣webfile[.w]␣[{changefile[.ch]|-}␣[outfile[.c]]]\n",
        "");
  else
    fatal("!␣Usage:␣cweave␣[options]␣webfile[.w]␣[{changefile[.ch]|-}␣[outfile[.tex]]]\n",
        "");
}
```

This code is used in section 70.

76. ⟨ Complain about argument length 76 ⟩ ≡
fatal("!␣Filename␣too␣long\n", ∗*argv*);
This code is used in sections 71, 72, and 73.

77. Output. Here is the code that opens the output file:

```
⟨ Definitions that should agree with CTANGLE and CWEAVE 2 ⟩ +≡
  FILE *C_file;  /* where output of CTANGLE goes */
  FILE *tex_file;  /* where output of CWEAVE goes */
  FILE *idx_file;  /* where index from CWEAVE goes */
  FILE *scn_file;  /* where list of sections from CWEAVE goes */
  FILE *active_file;  /* currently active file for CWEAVE output */
```

78.
```
⟨ Scan arguments and open output files 78 ⟩ ≡
  scan_args();
  if (program ≡ ctangle) {
    if ((C_file ← fopen(C_file_name, "w")) ≡ Λ) fatal("! Cannot open output file ", C_file_name);
  }
  else {
    if ((tex_file ← fopen(tex_file_name, "w")) ≡ Λ) fatal("! Cannot open output file ", tex_file_name);
  }
```
This code is used in section 4.

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

```
#define update_terminal fflush(stdout)  /* empty the terminal output buffer */
```

80. Terminal output uses *putchar* and *putc* when we have to translate from CWEB's code into the external character code, and *printf* when we just want to print strings. Several macros make other kinds of output convenient.

```
#define new_line putchar('\n')
#define putxchar putchar
#define term_write(a, b) fflush(stdout), fwrite(a, sizeof(char), b, stdout)
#define C_printf(c, a) fprintf(C_file, c, a)
#define C_putc(c) putc(c, C_file)  /* isn't C wonderfully consistent? */
```

81. We predeclare several standard system functions here instead of including their system header files, because the names of the header files are not as standard as the names of the functions. (For example, some C environments have `<string.h>` where others have `<strings.h>`.)

```
⟨ Predeclaration of procedures 33 ⟩ +≡
  extern int strlen();  /* length of string */
  extern int strcmp();  /* compare strings lexicographically */
  extern char *strcpy();  /* copy one string to another */
  extern int strncmp();  /* compare up to n string characters */
  extern char *strncpy();  /* copy up to n string characters */
```

82. Index.

Common code for CTANGLE and CWEAVE

(Version 3.64)