1. Introduction. This file contains the program `wmerge`, which takes two or more files and merges them according to the conventions of `CWEB`. Namely, it takes an ordinary `.w` file and and optional `.ch` file and sends the corresponding `.w`-style file to standard output (or to a named file), expanding all "includes" that might be specified by `@i` in the original `.w` file. (A more precise description appears in the section on "command line arguments" below.)

```
#include <stdio.h>
#include <stdlib.h>     /* declaration of getenv */
#include <ctype.h>      /* definition of isalpha, isdigit and so on */
  ⟨ Definitions 2 ⟩
  ⟨ Predeclarations of functions 3 ⟩
  ⟨ Functions 6 ⟩
  main(ac, av)
      int ac;
      char **av;
  {
    argc = ac;
    argv = av;
    ⟨ Set the default options 31 ⟩;
    ⟨ Scan arguments and open output file 41 ⟩;
    reset_input( );
    while (get_line( )) put_line( );
    fflush(out_file);
    check_complete( );
    fflush(out_file);
    return wrap_up( );
  }
```

2. ⟨ Definitions 2 ⟩ ≡

```
  typedef short boolean;
  typedef unsigned char eight_bits;
  typedef char ASCII;     /* type of characters inside WEB */
```

See also sections 5, 7, 8, 23, 30, and 40.

This code is used in section 1.

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

⟨ Predeclarations of functions 3 ⟩ ≡

```
  extern size_t strlen( );     /* length of string */
  extern char *strcpy( );      /* copy one string to another */
  extern int strncmp( );       /* compare up to n string characters */
  extern char *strncpy( );     /* copy up to n string characters */
```

See also sections 4, 24, and 32.

This code is used in section 1.

4. ⟨ Predeclarations of functions 3 ⟩ +≡

5. The lowest level of input to the WEB 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 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.

Some of the routines use the fact that it is safe to refer to ∗(*limit* + 2) without overstepping the bounds of the array.

```
#define buf_size 4096
⟨ Definitions 2 ⟩ +≡
  ASCII buffer[buf_size];      /* where each line of input goes */
  ASCII *buffer_end = buffer + buf_size − 2;      /* end of buffer */
  ASCII *limit;      /* points to the last character in the buffer */
  ASCII *loc;      /* points to the next character to be read from the buffer */
```

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

Incidentally, here's a curious fact about CWEB for those of you who are reading this file as an example of CWEB programming. The file `stdio.h` includes a typedef for the identifier **FILE**, which is not, strictly speaking, part of C. It turns out CWEAVE knows that **FILE** is a reserved word (after all, **FILE** is almost as common as **int**); indeed, CWEAVE knows all the types of the ISO standard C library. But if you're using other types like **caddr_t**, which is defined in `/usr/include/sys/types.h`, you should let WEAVE know that this is a type, either by including the `.h` file at WEB time (saying `@i /usr/include/sys/types.h`), or by using WEB's format command (saying `@f caddr_t int`). Either of these will make **caddr_t** be treated in the same way as **int**.

```
⟨ Functions 6 ⟩ ≡
  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);
  }
```

See also sections 9, 13, 15, 17, 22, 25, 28, and 33.

This code is used in section 1.

7. 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.

```
  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 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 */
```

8. 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))
⟨ Definitions 2 ⟩ +≡
  char change_buffer[buf_size];    /* next line of change_file */
  char *change_limit;    /* points to the last character in change_buffer */
```

9. 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.

```
⟨ Functions 6 ⟩ +≡
  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 10 ⟩;
    ⟨ Skip to the next nonblank line; return if end of file 11 ⟩;
    ⟨ Move buffer and limit to change_buffer and change_limit 12 ⟩;
  }
```

10. 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 10 ⟩ ≡
  while (1) {
    change_line++;
    if (¬input_ln(change_file)) return;
    if (limit < buffer + 2) continue;
    if (buffer[0] ≠ '@') continue;
    if (isupper(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 9.

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

```
⟨ Skip to the next nonblank line; return if end of file 11 ⟩ ≡
  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 9.

12. ⟨ Move *buffer* and *limit* to *change_buffer* and *change_limit* 12 ⟩ ≡

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

This code is used in sections 9 and 13.

13. 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*.

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

```
⟨ Functions 6 ⟩ +≡
  void check_change( )      /* switches to change_file if the buffers match */
  {
    int n = 0;      /* the number of discrepancies found */
    if (lines_dont_match) return;
    while (1) {
      changing = 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 = isupper(buffer[1]) ? tolower(buffer[1]) : buffer[1];
        ⟨ If the current line starts with @y, report any discrepancies and return 14 ⟩;
      }
      ⟨ Move buffer and limit to change_buffer and change_limit 12 ⟩;
      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++;
    }
  }
```

14. ⟨ If the current line starts with `@y`, report any discrepancies and **return** 14 ⟩ ≡

```
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;
    fprintf(stderr, "\n! Hmm... %d ", n);
    err_print("of the preceding lines failed to match");
  }
  change_depth = include_depth;
  return;
}
```

This code is used in section 13.

15. The *reset_input* procedure gets the program ready to read the user's `WEB` input.

⟨ Functions 6 ⟩ +≡

```
void reset_input( )
{
  limit = buffer;
  loc = buffer + 1;
  buffer[0] = ' ';
  ⟨ Open input files 16 ⟩;
  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;
}
```

16. The following code opens the input files.

⟨ Open input files 16 ⟩ ≡

```
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 15.

17. 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.

```
⟨ Functions 6 ⟩ +≡
  int get_line( )      /* inputs the next line */
  {
  restart:
    if (changing ∧ include_depth ≡ change_depth)
      ⟨ Read from change_file and maybe turn off changing 21 ⟩;
    if (¬changing ∨ include_depth > change_depth) {
      ⟨ Read from cur_file and maybe turn on changing 20 ⟩;
      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 19 ⟩;
    }
    return 1;
  }
  void put_line( )
  {
    char *ptr = buffer;
    while (ptr < limit) putc(*ptr++, out_file);
    putc('\n', out_file);
  }
```

18. 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;
        }
```

19. ⟨ Try to open include file, abort push if unsuccessful, go to *restart* 19 ⟩ ≡

```
{
  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;
    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;
      goto restart;      /* success */
    }
  }
  include_depth--;
  err_print("! Cannot open include file");
  goto restart;
}
```

This code is used in section 17.

20. ⟨ Read from *cur_file* and maybe turn on *changing* 20 ⟩ ≡

```
{
  cur_line++;
  while (¬input_ln(cur_file)) {      /* pop the stack or quit */
    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 17.

21. ⟨ Read from *change_file* and maybe turn off *changing* 21 ⟩ ≡

```
{
  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 */
    *limit = '␣';
    if (buffer[0] ≡ '@') {
      if (isupper(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;
      }
    }
  }
}
```

This code is used in section 17.

22. 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*.

```
⟨ Functions 6 ⟩ +≡
  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");
    }
  }
```

23. 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 2 ⟩ +≡
  int history = spotless;  /* indicates how bad this run was */
```

24. 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 `"!"`.

The actual error indications are provided by a procedure called **error**.

```
⟨ Predeclarations of functions 3 ⟩ +≡
  void err_print( );
```

25.

```
⟨ Functions 6 ⟩ +≡
  void err_print(s)  /* prints '.' and location of error message */
      char *s;
  {
    char *k, *l;  /* pointers into buffer */
    fprintf(stderr, *s ≡ '!' ? "\n%s" : "%s", s);
    if (web_file_open) ⟨ Print error location based on input buffer 26 ⟩
    else putc('\n', stderr);
    update_terminal;
    mark_error;
  }
```

26. 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 26 ⟩ ≡
  {
    if (changing ∧ include_depth ≡ change_depth)
      fprintf(stderr, ". (l. %d of change file)\n", change_line);
    else if (include_depth ≡ 0) fprintf(stderr, ". (l. %d)\n", cur_line);
    else fprintf(stderr, ". (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') putc(' ', stderr);
        else putc(∗k, stderr);      /∗ print the characters already read ∗/
      putc('\n', stderr);
      for (k = buffer; k < l; k++) putc(' ', stderr);      /∗ space out the next line ∗/
    }
    for (k = l; k < limit; k++) putc(∗k, stderr);      /∗ print the part not yet read ∗/
    putc('\n', stderr);
  }
```

This code is used in section 25.

27. 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.

```
#define fatal(s, t)
        {
          fprintf(stderr, s);
          err_print(t);
          history = fatal_message;
          exit(wrap_up( ));
        }
```

28. 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.

```
⟨ Functions 6 ⟩ +≡
  wrap_up( )
  {
    ⟨ Print the job history 29 ⟩;
    if (history > harmless_message) return (1);
    else return (0);
  }
```

29. ⟨ Print the job *history* 29 ⟩ ≡

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

This code is used in section 28.

30. Command line arguments. The user calls `wmerge` 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_happiness flags['h']    /* should lack of errors be announced? */
```

⟨ Definitions 2 ⟩ +≡

```
int argc;    /* copy of ac parameter to main */
char **argv;    /* copy of av parameter to main */
char out_file_name[max_file_name_length];    /* name of out_file */
boolean flags[128];    /* an option for each 7-bit code */
```

31. The *flags* will be initially 1.

⟨ Set the default options 31 ⟩ ≡

```
show_banner = show_happiness = 1;
```

This code is used in section 1.

32. We now must look at the command line arguments and set the file names accordingly. At least one file name must be present: the `WEB` file. It may have an extension, or it may omit it to get `'.w'` added.

If there is another 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.

⟨ Predeclarations of functions 3 ⟩ +≡

```
void scan_args();
```

33.

```
⟨ Functions 6 ⟩ +≡
  void scan_args( )
  {
    char *dot_pos;      /* position of '.' in the argument */
    register char *s;      /* register for scanning strings */
    boolean found_web = 0, found_change = 0, found_out = 0;
      /* have these names have been seen? */
    boolean flag_change;
    while (--argc > 0) {
      if (**(++argv) ≡ '-' ∨ **argv ≡ '+') ⟨ Handle flag argument 37 ⟩
      else {
        s = *argv; dot_pos = Λ;
        while (*s) {
          if (*s ≡ '.') dot_pos = s++;
          else if (*s ≡ '/') dot_pos = Λ, ++s;
          else s++;
        }
        if (¬found_web) ⟨ Make web_file_name 34 ⟩
        else if (¬found_change) ⟨ Make change_file_name from fname 35 ⟩
        else if (¬found_out) ⟨ Override output file name 36 ⟩
        else ⟨ Print usage error message and quit 38 ⟩;
      }
    }
    if (¬found_web) ⟨ Print usage error message and quit 38 ⟩;
    if (¬found_change) strcpy(change_file_name, "/dev/null");
  }
```

34. 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 34 ⟩ ≡
  {
    if (s − *argv > max_file_name_length − 5) ⟨ Complain about argument length 39 ⟩;
    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);
    *out_file_name = '\0';      /* this will print to stdout */
    found_web = 1;
  }
```

This code is used in section 33.

35. ⟨ Make *change_file_name* from *fname* 35 ⟩ ≡

```
{
  if (s − ∗argv > max_file_name_length − 4) ⟨ Complain about argument length 39 ⟩;
  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 33.

36. ⟨ Override output file name 36 ⟩ ≡

```
{
  if (s − ∗argv > max_file_name_length − 5) ⟨ Complain about argument length 39 ⟩;
  if (dot_pos ≡ Λ) sprintf(out_file_name, "%s.out", ∗argv);
  else strcpy(out_file_name, ∗argv);
  found_out = 1;
}
```

This code is used in section 33.

37. ⟨ Handle flag argument 37 ⟩ ≡

```
{
  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 33.

38. ⟨ Print usage error message and quit 38 ⟩ ≡

```
{
  fatal("! Usage: wmerge webfile[.w] [changefile[.ch] [outfile[.out]]]\n", "")
}
```

This code is used in section 33.

39. ⟨ Complain about argument length 39 ⟩ ≡

```
fatal("! Filename too long\n", ∗argv);
```

This code is used in sections 34, 35, and 36.

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

```
⟨ Definitions 2 ⟩ +≡
  FILE *out_file;      /* where output goes */
```

41. ⟨ Scan arguments and open output file 41 ⟩ ≡

```
  scan_args( );
  if (out_file_name[0] ≡ '\0') out_file = stdout;
  else if ((out_file = fopen(out_file_name, "w")) ≡ Λ)
    fatal("! Cannot open output file ", out_file_name);
```

This code is used in section 1.

42. 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(stderr)      /* empty the terminal output buffer */
```

43. Index.

ac: 1, 30.
alt_web_file_name: 7, 16, 34.
argc: 1, 30, 33.
argv: 1, 30, 33, 34, 35, 36, 37, 39.
ASCII: 2, 5.
av: 1, 30.
boolean: 2, 7, 30, 33.
buf_size: 5, 8.
buffer: 5, 6, 8, 10, 11, 12, 13, 14, 15, 17, 20, 21, 22, 25, 26.
buffer_end: 5, 6.
c: 6.
caddr_t: 6.
`Cannot open change file`: 16.
`Cannot open input file`: 16.
`Cannot open output file`: 41.
`Change file ended...`: 11, 13, 21.
`Change file entry did not match`: 22.
change_buffer: 8, 9, 12, 13, 20, 22.
change_depth: 7, 14, 15, 17, 20, 22, 26.
change_file: 7, 8, 10, 11, 13, 16, 21, 26.
change_file_name: 7, 16, 33, 35.
change_limit: 8, 9, 12, 13, 20, 22.
change_line: 7, 10, 11, 13, 15, 21, 26.
changing: 7, 8, 9, 13, 15, 17, 20, 21, 22, 26.
check_change: 13, 20.
check_complete: 1, 22.
cur_file: 7, 8, 13, 18, 19, 20, 26.
cur_file_name: 7, 19, 26.
cur_file_name_end: 19.
cur_line: 7, 13, 15, 19, 20, 26.
`CWEB file ended...`: 13.
`CWEBINPUTS`: 19.
dot_pos: 33, 34, 35, 36, 37.
eight_bits: 2.
`EOF`: 6.
err_print: 6, 10, 11, 13, 14, 17, 18, 19, 21, 22, 24, 25, 27.
error_message: 23, 29.
exit: 27.
fatal: 16, 27, 38, 39, 41.
fatal_message: 23, 27, 29.
fclose: 20.
feof: 6.
fflush: 1, 42.
file: 7.
file_name: 7.
flag_change: 33, 37.
flags: 30, 31, 37.
fopen: 16, 19, 41.
found_change: 33, 35.
found_out: 33, 36.
found_web: 33, 34.
fp: 6.
fprintf: 14, 25, 26, 27, 29.
get_line: 1, 17.
getc: 6.
getenv: 1, 19.
harmless_message: 23, 28, 29.
history: 23, 27, 28, 29.
`Hmm... n of the preceding...`: 14.
`Include file name ...`: 17, 19.
include_depth: 7, 13, 14, 15, 17, 18, 19, 20, 22, 26.
`Input line too long`: 6.
input_has_ended: 7, 13, 15, 17, 20.
input_ln: 5, 6, 10, 11, 13, 20, 21.
isalpha: 1.
isdigit: 1.
isupper: 10, 13, 21.
k: 6, 19, 25.
kk: 19.
l: 19, 25.
limit: 5, 6, 8, 10, 11, 12, 13, 15, 17, 19, 20, 21, 22, 26.
line: 7.
lines_dont_match: 8, 13.
loc: 5, 6, 10, 14, 15, 17, 19, 21, 22, 26.
main: 1, 30.
mark_error: 23, 25.
mark_harmless: 23.
max_file_name_length: 7, 19, 30, 34, 35, 36.
max_include_depth: 7, 17.
`Missing @x...`: 10.
n: 13.
out_file: 1, 17, 30, 40, 41.
out_file_name: 30, 34, 36, 41.
prime_the_change_buffer: 9, 15, 21.
ptr: 17.
put_line: 1, 17.
putc: 17, 25, 26.
reset_input: 1, 15.
restart: 17, 18, 19.
s: 25, 33.
scan_args: 32, 33, 41.
show_banner: 30, 31.
show_happiness: 29, 30, 31.
spotless: 23, 29.
sprintf: 34, 35, 36.
stderr: 14, 25, 26, 27, 29, 42.
stdout: 41.
strcpy: 3, 16, 19, 33, 34, 35, 36.
strlen: 3, 19.

strncmp: 3, 8.
strncpy: 3, 12, 22.
system dependencies: 6, 16, 26, 28, 32, 40, 42.
temp_file_name: 19.
tolower: 10, 13, 21.
`Too many nested includes`: 17.
too_long: 18, 19.
ungetc: 6.
update_terminal: 25, 42.
web_file: 7, 16, 22.
web_file_name: 7, 16, 34.
web_file_open: 7, 16, 25.
`Where is the match...`: 14, 21.
wrap_up: 1, 27, 28.
xyz_code: 13, 14.

⟨ Complain about argument length 39 ⟩ Used in sections 34, 35, and 36.
⟨ Definitions 2, 5, 7, 8, 23, 30, 40 ⟩ Used in section 1.
⟨ Functions 6, 9, 13, 15, 17, 22, 25, 28, 33 ⟩ Used in section 1.
⟨ Handle flag argument 37 ⟩ Used in section 33.
⟨ If the current line starts with `@y`, report any discrepancies and **return** 14 ⟩ Used in section 13.
⟨ Make *change_file_name* from *fname* 35 ⟩ Used in section 33.
⟨ Make *web_file_name* 34 ⟩ Used in section 33.
⟨ Move *buffer* and *limit* to *change_buffer* and *change_limit* 12 ⟩ Used in sections 9 and 13.
⟨ Open input files 16 ⟩ Used in section 15.
⟨ Override output file name 36 ⟩ Used in section 33.
⟨ Predeclarations of functions 3, 4, 24, 32 ⟩ Used in section 1.
⟨ Print error location based on input buffer 26 ⟩ Used in section 25.
⟨ Print the job *history* 29 ⟩ Used in section 28.
⟨ Print usage error message and quit 38 ⟩ Used in section 33.
⟨ Read from *change_file* and maybe turn off *changing* 21 ⟩ Used in section 17.
⟨ Read from *cur_file* and maybe turn on *changing* 20 ⟩ Used in section 17.
⟨ Scan arguments and open output file 41 ⟩ Used in section 1.
⟨ Set the default options 31 ⟩ Used in section 1.
⟨ Skip over comment lines in the change file; **return** if end of file 10 ⟩ Used in section 9.
⟨ Skip to the next nonblank line; **return** if end of file 11 ⟩ Used in section 9.
⟨ Try to open include file, abort push if unsuccessful, go to *restart* 19 ⟩ Used in section 17.

WMERGE

	Section	Page
Introduction	1	1
Reporting errors to the user	23	12
Command line arguments	30	15
Output	40	18
Index	43	19