December 28, 2021 at 16:15

1. An example of CWEB. This example, based on a program by Klaus Guntermann and Joachim Schrod [*TUGboat* **7** (1986), 134–137] presents the "word count" program from UNIX, rewritten in CWEB to demonstrate literate programming in C. The level of detail in this document is intentionally high, for didactic purposes; many of the things spelled out here don't need to be explained in other programs.

The purpose of `wc` is to count lines, words, and/or characters in a list of files. The number of lines in a file is the number of newline characters it contains. The number of characters is the file length in bytes. A "word" is a maximal sequence of consecutive characters other than newline, space, or tab, containing at least one visible ASCII code. (We assume that the standard ASCII code is in use.)

This version of `wc` has a nonstandard "silent" option (`-s`), which suppresses printing except for the grand totals over all files.

2. Most CWEB programs share a common structure. It's probably a good idea to state the overall structure explicitly at the outset, even though the various parts could all be introduced in unnamed sections of the code if we wanted to add them piecemeal.

Here, then, is an overview of the file `wc.c` that is defined by this CWEB program `wc.w`:

⟨ Header files to include 4 ⟩
⟨ Global variables 5 ⟩
⟨ Prototypes 3 ⟩
⟨ Functions 23 ⟩
⟨ The main program 7 ⟩

3.

⟨ Prototypes 3 ⟩ ≡
```
void wc_print(char *which, long char_count, long word_count, long line_count);
```
This code is used in section 2.

4. We must include the standard I/O definitions, since we want to send formatted output to *stdout* and *stderr*.

⟨ Header files to include 4 ⟩ ≡
```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <fcntl.h>
#include <unistd.h>
```
This code is used in section 2.

5. The *status* variable will tell the operating system if the run was successful or not, and *prog_name* is used in case there's an error message to be printed.

```
#define OK 0 /* status code for successful run */
#define usage_error 1 /* status code for improper syntax */
#define cannot_open_file 2 /* status code for file access error */
```
⟨ Global variables 5 ⟩ ≡
```
int status = OK; /* exit status of command, initially OK */
char *prog_name; /* who we are */
```
See also section 16.

This code is used in section 2.

6. Now we come to the general layout of the *main* function.

7. Argument *argc* is the number of arguments on the UNIX command line and *argv* are the arguments themselves, an array of strings.

```
⟨ The main program 7 ⟩ ≡
  int main(int argc, char **argv)
  {
    ⟨ Variables local to main 8 ⟩
    prog_name = argv[0];
    ⟨ Set up option selection 9 ⟩;
    ⟨ Process all the files 10 ⟩;
    ⟨ Print the grand totals if there were multiple files 21 ⟩;
    exit(status);
  }
```

This code is used in section 2.

8. If the first argument begins with a '`-`', the user is choosing the desired counts and specifying the order in which they should be displayed. Each selection is given by the initial character (lines, words, or characters). For example, '`-cl`' would cause just the number of characters and the number of lines to be printed, in that order. The default, if no special argument is given, is '`-lwc`'.

We do not process this string now; we simply remember where it is. It will be used to control the formatting at output time.

If the '`-`' is immediately followed by '`s`', only summary totals are printed.

```
⟨ Variables local to main 8 ⟩ ≡
  int file_count;      /* how many files there are */
  char *which;         /* which counts to print */
  int silent = 0;      /* nonzero if the silent option was selected */
```

See also sections 11 and 14.

This code is used in section 7.

9.

```
⟨ Set up option selection 9 ⟩ ≡
  which = "lwc";       /* if no option is given, print all three values */
  if (argc > 1 ∧ *argv[1] ≡ '-') {
    argv[1]++;
    if (*argv[1] ≡ 's') silent = 1, argv[1]++;
    if (*argv[1]) which = argv[1];
    argc--;
    argv++;
  }
  file_count = argc - 1;
```

This code is used in section 7.

10. Now we scan the remaining arguments and try to open a file, if possible. The file is processed and its statistics are given. We use a **do** ... **while** loop because we should read from the standard input if no file name is given.

```
⟨ Process all the files 10 ⟩ ≡
  argc −−;
  do {
    ⟨ If a file is given, try to open ∗(++argv); continue if unsuccessful 12 ⟩;
    ⟨ Initialize pointers and counters 15 ⟩;
    ⟨ Scan file 17 ⟩;
    ⟨ Write statistics for file 19 ⟩;
    ⟨ Close file 13 ⟩;
    ⟨ Update grand totals 20 ⟩;      /∗ even if there is only one file ∗/
  } while (−−argc > 0);
```

This code is used in section 7.

11. Here's the code to open the file. A special trick allows us to handle input from *stdin* when no name is given. Recall that the file descriptor to *stdin* is 0; that's what we use as the default initial value.

```
⟨ Variables local to main 8 ⟩ +≡
  int fd = 0;      /∗ file descriptor, initialized to stdin ∗/
```

12. `#define READ_ONLY 0` /∗ read access code for system *open* routine ∗/

```
⟨ If a file is given, try to open ∗(++argv); continue if unsuccessful 12 ⟩ ≡
  if (file_count > 0 ∧ (fd = open(∗(++argv), READ_ONLY)) < 0) {
    fprintf(stderr, "%s: cannot open file %s\n", prog_name, ∗argv);
    status |= cannot_open_file;
    file_count −−;
    continue;
  }
```

This code is used in section 10.

13.

```
⟨ Close file 13 ⟩ ≡
  close(fd);
```

This code is used in section 10.

14. We will do some homemade buffering in order to speed things up: Characters will be read into the *buffer* array before we process them. To do this we set up appropriate pointers and counters.

```
#define buf_size BUFSIZ      /∗ stdio.h's BUFSIZ is chosen for efficiency ∗/
⟨ Variables local to main 8 ⟩ +≡
  char buffer[buf_size];      /∗ we read the input into this array ∗/
  register char ∗ptr;      /∗ the first unprocessed character in buffer ∗/
  register char ∗buf_end;      /∗ the first unused position in buffer ∗/
  register int c;      /∗ current character, or number of characters just read ∗/
  int in_word;      /∗ are we within a word? ∗/
  long word_count, line_count, char_count;
    /∗ number of words, lines, and characters found in the file so far ∗/
```

15.

```
⟨ Initialize pointers and counters 15 ⟩ ≡
  ptr = buf_end = buffer;
  line_count = word_count = char_count = 0;
  in_word = 0;
```

This code is used in section 10.

16. The grand totals must be initialized to zero at the beginning of the program. If we made these variables local to *main*, we would have to do this initialization explicitly; however, C's globals are automatically zeroed. (Or rather, "statically zeroed.") (Get it?)

```
⟨ Global variables 5 ⟩ +≡
  long tot_word_count, tot_line_count, tot_char_count;      /* total number of words, lines, and chars */
```

17. The present section, which does the counting that is `wc`'s *raison d'être*, was actually one of the simplest to write. We look at each character and change state if it begins or ends a word.

```
⟨ Scan file 17 ⟩ ≡
  while (1) {
    ⟨ Fill buffer if it is empty; break at end of file 18 ⟩;
    c = *ptr++;
    if (c > '␣' ∧ c < °177) {      /* visible ASCII codes */
      if (¬in_word) {
        word_count++;
        in_word = 1;
      }
      continue;
    }
    if (c ≡ '\n') line_count++;
    else if (c ≠ '␣' ∧ c ≠ '\t') continue;
    in_word = 0;      /* c is newline, space, or tab */
  }
```

This code is used in section 10.

18. Buffered I/O allows us to count the number of characters almost for free.

```
⟨ Fill buffer if it is empty; break at end of file 18 ⟩ ≡
  if (ptr ≥ buf_end) {
    ptr = buffer;
    c = read(fd, ptr, buf_size);
    if (c ≤ 0) break;
    char_count += c;
    buf_end = buffer + c;
  }
```

This code is used in section 17.

19. It's convenient to output the statistics by defining a new function *wc_print*; then the same function can be used for the totals. Additionally we must decide here if we know the name of the file we have processed or if it was just *stdin*.

```
⟨ Write statistics for file 19 ⟩ ≡
  if (¬silent) {
    wc_print(which, char_count, word_count, line_count);
    if (file_count) printf("␣%s\n", *argv);      /* not stdin */
    else printf("\n");      /* stdin */
  }
```

This code is used in section 10.

20. ⟨ Update grand totals 20 ⟩ ≡

```
tot_line_count += line_count;
tot_word_count += word_count;
tot_char_count += char_count;
```

This code is used in section 10.

21. We might as well improve a bit on UNIX's `wc` by displaying the number of files too.

⟨ Print the grand totals if there were multiple files 21 ⟩ ≡

```
if (file_count > 1 ∨ silent) {
  wc_print(which, tot_char_count, tot_word_count, tot_line_count);
  if (¬file_count) printf("\n");
  else printf("␣total␣in␣%d␣file%s\n", file_count, file_count > 1 ? "s" : "");
}
```

This code is used in section 7.

22. Here now is the function that prints the values according to the specified options. The calling routine is supposed to supply a newline. If an invalid option character is found we inform the user about proper usage of the command. Counts are printed in 8-digit fields so that they will line up in columns.

```
#define print_count(n) printf("%8ld", n)
```

23. Argument *which* tells which counts to print and *char_count*, *word_count*, *line_count* are the given totals.

⟨ Functions 23 ⟩ ≡

```
void wc_print(char *which, long char_count, long word_count, long line_count)
{
  while (*which)
    switch (*which++) {
    case 'l': print_count(line_count);
      break;
    case 'w': print_count(word_count);
      break;
    case 'c': print_count(char_count);
      break;
    default:
      if ((status & usage_error) ≡ 0) {
        fprintf(stderr, "\nUsage:␣%s␣[-lwc]␣[filename␣...]\n", prog_name);
        status |= usage_error;
      }
    }
}
```

This code is used in section 2.

24. Incidentally, a test of this program against the system `wc` command on a SPARCstation showed that the "official" `wc` was slightly slower. Furthermore, although that `wc` gave an appropriate error message for the options '`-abc`', it made no complaints about the options '`-labc`'! Dare we suggest that the system routine might have been better if its programmer had used a more literate approach?

25. Index. Here is a list of the identifiers used, and where they appear. Underlined entries indicate the place of definition. Error messages are also shown.

⟨ Close file 13 ⟩ Used in section 10.
⟨ Fill *buffer* if it is empty; **break** at end of file 18 ⟩ Used in section 17.
⟨ Functions 23 ⟩ Used in section 2.
⟨ Global variables 5, 16 ⟩ Used in section 2.
⟨ Header files to include 4 ⟩ Used in section 2.
⟨ If a file is given, try to open $*(++argv)$; **continue** if unsuccessful 12 ⟩ Used in section 10.
⟨ Initialize pointers and counters 15 ⟩ Used in section 10.
⟨ Print the grand totals if there were multiple files 21 ⟩ Used in section 7.
⟨ Process all the files 10 ⟩ Used in section 7.
⟨ Prototypes 3 ⟩ Used in section 2.
⟨ Scan file 17 ⟩ Used in section 10.
⟨ Set up option selection 9 ⟩ Used in section 7.
⟨ The main program 7 ⟩ Used in section 2.
⟨ Update grand totals 20 ⟩ Used in section 10.
⟨ Variables local to *main* 8, 11, 14 ⟩ Used in section 7.
⟨ Write statistics for file 19 ⟩ Used in section 10.