Counting words

Silvio Levy Donald E. Knuth

August 29, 1995

1. An example of CWEB

This example, based on a program by Klaus Guntermann and Joachim Schrod [1] presents the "word count" program from UNIX, rewritten in CWEB to demonstrate literate programming in C.[1] 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.)

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 `wcltx.c` that is defined by this CWEB program `wcltx.w`:

```
⟨ Header files to include 3 ⟩
⟨ Global variables 4 ⟩
⟨ Functions 20 ⟩
⟨ The main program 5 ⟩
```

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

```
⟨ Header files to include 3 ⟩ ≡
#include <stdio.h>
```

This code is used in chunk 2.

[*] LaTeX markup by Joachim Schrod.

[1] Incidentally, Klaus Guntermann and Joachim Schrod are now using this CWEB and have stopped the support for their version. Joachim Schrod contributed the LaTeX style used for the markup of this document.

4. 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 4 ⟩ ≡
  int status = OK;      /∗ exit status of command, initially OK ∗/
  char ∗prog_name;      /∗ who we are ∗/
```

See also chunk 14.

This code is used in chunk 2.

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

```
⟨ The main program 5 ⟩ ≡
  main(argc, argv)
      int argc;
         /∗ the number of arguments on the UNIX command line ∗/
      char ∗∗argv;      /∗ the arguments themselves, an array of strings ∗/
  {
    ⟨ Variables local to main 6 ⟩
    prog_name = argv[0];
    ⟨ Set up option selection 7 ⟩;
    ⟨ Process all the files 8 ⟩;
    ⟨ Print the grand totals if there were multiple files 19 ⟩;
    exit(status);
  }
```

This code is used in chunk 2.

6. 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.

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

```
⟨ Variables local to main 6 ⟩ ≡
  int file_count;      /∗ how many files there are ∗/
  char ∗which;      /∗ which counts to print ∗/
```

See also chunks 9 and 12.

This code is used in chunk 5.

7. ⟨ Set up option selection 7 ⟩ ≡

```
which = "lwc";      /* if no option is given, print all three values */
if (argc > 1 ∧ *argv[1] ≡ '-') {
   which = argv[1] + 1;
   argc−−;
   argv++;
}
file_count = argc − 1;
```

This code is used in chunk 5.

8. 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 8 ⟩ ≡

```
argc−−;
do {
   ⟨ If a file is given, try to open *(++argv); continue if unsuccessful 10 ⟩;
   ⟨ Initialize pointers and counters 13 ⟩;
   ⟨ Scan file 15 ⟩;
   ⟨ Write statistics for file 17 ⟩;
   ⟨ Close file 11 ⟩;
   ⟨ Update grand totals 18 ⟩;      /* even if there is only one file */
}  while (−−argc > 0);
```

This code is used in chunk 5.

9. 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* 6 ⟩ +≡

```
int fd = 0;      /* file descriptor, initialized to stdin */
```

10. **#define** READ_ONLY 0
/* read access code for system *open* routine */

⟨ If a file is given, try to open *(++*argv*); **continue** if unsuccessful 10 ⟩ ≡

```
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 chunk 8.

11. ⟨ Close file 11 ⟩ ≡

```
  close(fd);
```

This code is used in chunk 8.

12. 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 6 ⟩ +≡
  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 */
```

13. ⟨ Initialize pointers and counters 13 ⟩ ≡

```
  ptr = buf_end = buffer;
  line_count = word_count = char_count = 0;
  in_word = 0;
```

This code is used in chunk 8.

14. 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 4 ⟩ +≡
  long tot_word_count, tot_line_count, tot_char_count;
      /* total number of words, lines, and chars */
```

15. 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 15 ⟩ ≡
  while (1) {
    ⟨ Fill buffer if it is empty; break at end of file 16 ⟩;
    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 chunk 8.

16. 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 16 ⟩ ≡

```
  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 chunk 15.

17. 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 17 ⟩ ≡

```
  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 chunk 8.

18. ⟨ Update grand totals 18 ⟩ ≡

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

This code is used in chunk 8.

19. 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 19 ⟩ ≡
  if (file_count > 1) {
    wc_print(which, tot_char_count, tot_word_count, tot_line_count);
    printf("␣total␣in␣%d␣files\n", file_count);
  }
```

This code is used in chunk 5.

20. 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)

⟨ Functions 20 ⟩ ≡
  wc_print(which, char_count, word_count, line_count)
      char *which;      /* which counts to print */
      long char_count, word_count, line_count;      /* given totals */
  {
    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 chunk 2.

21. 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?

References

[1] Klaus Guntermann and Joachim Schrod. WEB adapted to C. *TUGboat*, 7(3):134–137, 1986.

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.

List of Refinements

⟨ Close file 11 ⟩ Used in chunk 8.
⟨ Fill *buffer* if it is empty; **break** at end of file 16 ⟩ Used in chunk 15.
⟨ Functions 20 ⟩ Used in chunk 2.
⟨ Global variables 4, 14 ⟩ Used in chunk 2.
⟨ Header files to include 3 ⟩ Used in chunk 2.
⟨ If a file is given, try to open $*(\mathord{+}\!\mathord{+}\,argv)$; **continue** if unsuccessful 10 ⟩ Used in chunk 8.
⟨ Initialize pointers and counters 13 ⟩ Used in chunk 8.
⟨ Print the grand totals if there were multiple files 19 ⟩ Used in chunk 5.
⟨ Process all the files 8 ⟩ Used in chunk 5.
⟨ Scan file 15 ⟩ Used in chunk 8.
⟨ Set up option selection 7 ⟩ Used in chunk 5.
⟨ The main program 5 ⟩ Used in chunk 2.
⟨ Update grand totals 18 ⟩ Used in chunk 8.
⟨ Variables local to *main* 6, 9, 12 ⟩ Used in chunk 5.
⟨ Write statistics for file 17 ⟩ Used in chunk 8.