An Example of noweb

Norman Ramsey
Dept. of Computer Science, Princeton University
Princeton, NJ 08544

Contents

Introduction

The following short program illustrates the use of noweb, a low-tech tool for literate programming. The purpose of the program is to provide a basis for comparing WEB and noweb, so I have used a program that has been published before; the text, code, and presentation are taken from Chapter 12 of D. E. Knuth, *Literate Programming* (volume 27 of *Center for the Study of Language and Information Lecture Notes*, Stanford Univ., 1992).

The notable differences are:

- When displaying source code, noweb uses different typography. In particular, WEB makes good use of multiple fonts and the ablity to typeset mathematics, and it may use mathematical symbols in place of C symbols (e.g. a logical ``and'' symbol for ``&&''). noweb uses a single fixed-width font for code.
- noweb can work with HTML, and I have used HTML in this example.
- noweb has no numbered ``sections.'' When cross-referencing is needed, noweb uses hypertext links or page numbers.
- noweb has no special support for macros. In the sample program, I have used a ``Definitions'' chunk to hold macro definitions.
- noweb's index of identifiers is less accurate than WEB's, because it uses a language-independent heuristic to find identifiers. This heuristic may erroneously find ``uses'' of identifiers in string literals or comments. Although noweb does have a language-dependent algorithm for finding definitions of identifiers, that algorithm is less reliable than CWEB's, because noweb does not really parse C code.
- The CWEB version of this program has semicolons following most uses of <...>. WEB needs the semicolon or its equivalent to make its prettyprinting come out right. Because it does not attempt prettyprinting, noweb needs no semicolons.

Counting words

This example, based on a program by Klaus Guntermann and Joachim Schrod (`WEB adapted to C.' *TUGboat* **7**(3):134-7, Oct. 1986) and a program by Silvio Levy and D. E. Knuth (Ch. 12 of *Literate Programming*), presents the ``word count'' program from Unix, rewritten in noweb to demonstrate literate programming using noweb. 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.)

Most literate C 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 chunks named <*> if we wanted to add them piecemeal.

Here, then, is an overview of the file wc.c that is defined by the noweb program wc.nw:

```
⟨*⟩=
⟨Header files to include⟩
⟨Definitions⟩
⟨Global variables⟩
⟨Functions⟩
⟨The main program⟩
```

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

```
⟨Header files to include⟩= (←U)
#include <stdio.h>
```

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.

```
⟨Definitions⟩= (←U) [D→]
#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 */
```

Defines cannot_open_file, OK, usage_error (links are to index).

```
⟨Global variables⟩= (←U) [D→]
int status = OK;
  /* exit status of command, initially OK */
char *prog_name;
  /* who we are */
```

Defines prog_name, status (links are to index).

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

```
⟨The main program⟩= (←U)
main(argc, argv)
  int argc;
    /* number of arguments on UNIX command line */
  char **argv;
    /* the arguments, an array of strings */
{
  ⟨Variables local to main⟩
  prog_name = argv[0];
  ⟨Set up option selection⟩
```

```
  ⟨Process all the files⟩
  ⟨Print the grand totals if there were multiple files⟩
  exit(status);
}
```

Defines argc, argv, main (links are to index).

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⟩= (<-U) [D->]
int file_count;
  /* how many files there are */
char *which;
  /* which counts to print */
```

Defines file_count, which (links are to index).

```
⟨Set up option selection⟩= (<-U)
which = "lwc";
  /* if no option is given, print 3 values */
if (argc > 1 && *argv[1] == '-') {
  which = argv[1] + 1;
  argc--;
  argv++;
}
file_count = argc - 1;
```

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

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⟩+= (<-U) [<-D->]
int fd = 0;
  /* file descriptor, initialized to stdin */
```

Defines fd (links are to index).

```
⟨Definitions⟩+= (<-U) [<-D->]
#define READ_ONLY 0
  /* read access code for system open */
```

Defines READ_ONLY (links are to index).

```
⟨If a file is given, try to open *(++argv); continue if unsuccessful⟩= (⟨-U)
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;
}
```

```
⟨Close file⟩= (⟨-U)
close(fd);
```

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.

```
⟨Definitions⟩+= (⟨-U) [⟨-D->]
#define buf_size BUFSIZ
  /* stdio.h BUFSIZ chosen for efficiency */
```

Defines buf_size (links are to index).

```
⟨Variables local to main⟩+= (⟨-U) [⟨-D]
char buffer[buf_size];
  /* we read the input into this array */
register char *ptr;
  /* first unprocessed character in buffer */
register char *buf_end;
  /* the first unused position in buffer */
register int c;
  /* current char, or # of chars just read */
int in_word;
  /* are we within a word? */
long word_count, line_count, char_count;
  /* # of words, lines, and chars so far */
```

Defines buf_end, buffer, c, char_count, in_word, line_count, ptr, word_count (links are to index).

```
⟨Initialize pointers and counters⟩= (⟨-U)
ptr = buf_end = buffer;
line_count = word_count = char_count = 0;
in_word = 0;
```

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⟩+= (⟨-U) [⟨-D]
long tot_word_count, tot_line_count,
     tot_char_count;
  /* total number of words, lines, chars */
```

Defines tot_line_count, tot_word_count (links are to index).

The present chunk, which does the counting that is wc's *raison d'etre*, 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⟩= (<-U)
while (1) {
  ⟨Fill buffer if it is empty; break at end of file⟩
  c = *ptr++;
  if (c > ' ' && c < 0177) {
    /* 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 */
}
```

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⟩= (<-U)
if (ptr >= buf end) {
  ptr = buffer;
  c = read(fd, ptr, buf size);
  if (c <= 0) break;
  char count += c;
  buf end = buffer + c;
}
```

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⟩= (<-U)
wc print(which, char count, word count,
         line count);
if (file count)
  printf(" %s\n", *argv); /* not stdin */
else
  printf("\n");           /* stdin */
```

Defines `wc print` (links are to index).

```
⟨Update grand totals⟩= (<-U)
tot line count += line count;
tot word count += word count;
tot_char_count += char count;
```

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⟩= (<-U)
if (file count > 1) {
  wc print(which, tot_char_count,
           tot word count, tot line count);
  printf(" total in %d files\n", file count);
}
```

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.

```
⟨Definitions⟩+= (<-U) [<-D]
#define print count(n) printf("%8ld", n)
```

Defines print count (links are to index).

```
<Functions>= (<-U)
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;
        }
    }
}
```

Defines wc_print (links are to index).

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?

Index

Chunks

- *<Update grand totals>*: U1, D2
- *<Variables local to `main`>*: U1, D2, D3, D4
- *<Write statistics for file>*: U1, D2

Indentifiers