1 An example of noweb

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 [2, Chapter 12]. 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. "∧" for "&&"). noweb uses a single fixed-width font for code.
- noweb can work with LaTeX, and I have used LaTeX in this example.
- noweb has no numbered "sections." When numbers are needed for cross-referencing, noweb uses page numbers. If two or more chunks appear on a page, for example, page 24, they are distinguished by appending a letter to the page number, for example, 24a or 24b.
- noweb has no special support for macros. In the sample program, I have used the chunk "⟨*Definitions* ??⟩" to hold macro definitions.
- noweb does not recognize C identifier definitions automatically, so I had to add a list of defined identifiers to each code chunk. Because noweb is language-independent, it must use a heuristic to find uses of identifiers. This heuristic can be fooled into finding false "uses" in comments or string literals, such as the use of status in chunk ??.
- 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.
- Both WEB and noweb write chunk cross-reference information in footnote font below each code chunk, for example, "" Unlike WEB, noweb also includes cross-reference information for identifiers, for example, "Defines file_count, used in chunks 4a, 5a, and 7." This information is generated using the @ %def markings in the noweb source.

1.1 Counting words

This example, based on a program by Klaus Guntermann and Joachim Schrod [1] and a program by Silvio Levy and D. E. Knuth [2, Chapter 12], 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`:

2a ⟨* 2a⟩≡
⟨*Header files to include* 2b⟩
⟨*Definitions* 2c⟩
⟨*Global variables* 3a⟩
⟨*Functions* 8b⟩
⟨*The main program* 3b⟩

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

2b ⟨*Header files to include* 2b⟩≡ (2a)
```
#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.

2c ⟨*Definitions* 2c⟩≡ (2a) 4d▷
```
#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`, used in chunk 5a.
`OK`, used in chunk 3a.
`usage_error`, used in chunk 8b.
Uses `status` 3a.

3a ⟨*Global variables* 3a⟩≡ (2a) 6b▷
```
int status = OK;
  /* exit status of command, initially OK */
char *prog_name;
  /* who we are */
```
Defines:
`prog_name`, used in chunks 3b, 5a, and 8b.
`status`, used in chunks 2c, 3b, 5a, and 8b.
Uses `OK` 2c.

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

3b ⟨*The main program* 3b⟩≡ (2a)
```
main(argc, argv)
  int argc;
    /* number of arguments on UNIX command line */
  char **argv;
    /* the arguments, an array of strings */
{
  ⟨Variables local to main 3c⟩
  prog_name = argv[0];
  ⟨Set up option selection 4a⟩
  ⟨Process all the files 4b⟩
  ⟨Print the grand totals if there were multiple files 7d⟩
  exit(status);
}
```
Defines:
`argc`, never used.
`argv`, never used.
Uses `prog_name` 3a and `status` 3a.

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.

3c ⟨*Variables local to* `main` 3c⟩≡ (3b) 4c▷
```
int file_count;
  /* how many files there are */
char *which;
  /* which counts to print */
```
Defines:
`file_count`, used in chunks 4a, 5a, and 7.
`which`, used in chunks 4a, 7, and 8b.

4a ⟨*Set up option selection* 4a⟩≡ (3b)

```
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;
```

Uses `file_count` 3c and `which` 3c.

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.

4b ⟨*Process all the files* 4b⟩≡ (3b)

```
argc--;
do {
  ⟨If a file is given, try to open *(++argv); continue if unsuccessful 5a⟩
  ⟨Initialize pointers and counters 6a⟩
  ⟨Scan file 6c⟩
  ⟨Write statistics for file 7b⟩
  ⟨Close file 5b⟩
  ⟨Update grand totals 7c⟩
    /* 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.

4c ⟨*Variables local to* `main` 3c⟩+≡ (3b) ◁3c 5d▷

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

Defines:
`fd`, never used.

4d ⟨*Definitions* 2c⟩+≡ (2a) ◁2c 5c▷

```
#define READ_ONLY 0
  /* read access code for system open */
```

Defines:
`READ_ONLY`, used in chunk 5a.

5a ⟨*If a file is given, try to open* `*(++argv); continue` *if unsuccessful* 5a⟩≡ (4b)

```
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;
}
```

Uses `cannot_open_file` 2c, `file_count` 3c, `prog_name` 3a, `READ_ONLY` 4d, and `status` 3a.

5b ⟨*Close file* 5b⟩≡ (4b)

```
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.

5c ⟨*Definitions* 2c⟩+≡ (2a) ◁4d 8a▷

```
#define buf_size BUFSIZ
  /* stdio.h BUFSIZ chosen for efficiency */
```

Defines:
`buf_size`, used in chunks 5d and 7a.

5d ⟨*Variables local to* `main` 3c⟩+≡ (3b) ◁4c

```
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`, used in chunks 6a and 7a.
`buffer`, used in chunks 6a and 7a.
`c`, used in chunks 6c and 7a.
`char_count`, used in chunks 6–8.
`in_word`, used in chunk 6.
`line_count`, used in chunks 6–8.
`ptr`, used in chunks 6 and 7a.
`word_count`, used in chunks 6–8.
Uses `buf_size` 5c.

6a ⟨*Initialize pointers and counters* 6a⟩≡ (4b)

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

Uses buf_end 5d, buffer 5d, char_count 5d, in_word 5d, line_count 5d, ptr 5d, and word_count 5d.

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?)

6b ⟨*Global variables* 3a⟩+≡ (2a) ◁3a

```
long tot_word_count, tot_line_count,
     tot_char_count;
  /* total number of words, lines, chars */
```

Defines:
tot_line_count, used in chunk 7.
tot_word_count, used in chunk 7.

The present chunk, 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.

6c ⟨*Scan file* 6c⟩≡ (4b)

```
while (1) {
  ⟨Fill buffer if it is empty; break at end of file 7a⟩
  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 */
}
```

Uses c 5d, in_word 5d, line_count 5d, ptr 5d, and word_count 5d.

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

7a ⟨*Fill* `buffer` *if it is empty;* `break` *at end of file* 7a⟩≡ (6c)

```
if (ptr >= buf_end) {
  ptr = buffer;
  c = read(fd, ptr, buf_size);
  if (c <= 0) break;
  char_count += c;
  buf_end = buffer + c;
}
```

Uses `buf_end` 5d, `buf_size` 5c, `buffer` 5d, `c` 5d, `char_count` 5d, and `ptr` 5d.

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`.

7b ⟨*Write statistics for file* 7b⟩≡ (4b)

```
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`, used in chunks 7d and 8b.
Uses `char_count` 5d, `file_count` 3c, `line_count` 5d, `which` 3c, and `word_count` 5d.

7c ⟨*Update grand totals* 7c⟩≡ (4b)

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

Uses `char_count` 5d, `line_count` 5d, `tot_line_count` 6b, `tot_word_count` 6b, and `word_count` 5d.

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

7d ⟨*Print the grand totals if there were multiple files* 7d⟩≡ (3b)

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

Uses `file_count` 3c, `tot_line_count` 6b, `tot_word_count` 6b, `wc_print` 7b, and `which` 3c.

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.

8a ⟨*Definitions* 2c⟩+≡ (2a) ◁5c

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

Defines:
`print_count`, used in chunk 8b.

8b ⟨*Functions* 8b⟩≡ (2a)

```
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;
        }
    }
}
```

Uses `char_count` 5d, `line_count` 5d, `print_count` 8a, `prog_name` 3a, `status` 3a, `usage_error` 2c, `wc_print` 7b, `which` 3c, and `word_count` 5d.

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?

List of code chunks

This list is generated automatically. The numeral is that of the first definition of the chunk.

⟨ * 2a ⟩
⟨ *Close file* 5b ⟩
⟨ *Definitions* 2c ⟩
⟨ *Fill* `buffer` *if it is empty;* `break` *at end of file* 7a ⟩
⟨ *Functions* 8b ⟩
⟨ *Global variables* 3a ⟩
⟨ *Header files to include* 2b ⟩
⟨ *If a file is given, try to open* `*(++argv);` `continue` *if unsuccessful* 5a ⟩
⟨ *Initialize pointers and counters* 6a ⟩
⟨ *Print the grand totals if there were multiple files* 7d ⟩
⟨ *Process all the files* 4b ⟩
⟨ *Scan file* 6c ⟩
⟨ *Set up option selection* 4a ⟩
⟨ *The main program* 3b ⟩
⟨ *Update grand totals* 7c ⟩
⟨ *Variables local to* `main` 3c ⟩
⟨ *Write statistics for file* 7b ⟩

Index

Here is a list of the identifiers used, and where they appear. Underlined entries indicate the place of definition. This index is generated automatically.

References

[1] Klaus Guntermann and Joachim Schrod. WEB adapted to C. *TUGboat*, 7(3):134–137, October 1986. 1.1

[2] Donald E. Knuth. *Literate Programming*, volume 27 of *Center for the Study of Language and Information Lecture Notes*. Leland Stanford Junior University, Stanford, California, 1992. 1, 1.1