

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 [?, 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 ability to typeset mathematics, and it may use mathematical symbols in place of C symbols (e.g. “ \wedge ” for “`&&`”). `noweb` uses a single fixed-width font for code.
- `noweb` can work with L^AT_EX, and I have used L^AT_EX 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 2c*” 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 `2c`.
- 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_countfile:uncount`” 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 [?] and a program by Silvio Levy and D. E. Knuth [?, 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 2d>
    <Functions 7c>
    <The main program 3a>
```

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>≡
    #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>≡
    #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 */
```

```
2d <Global variables 2d>≡
    int status = OK;
        /* exit status of command, initially OK */
    char *prog_name;
        /* who we are */
```

Defines:
`stdout`, used in chunk 2d.

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

```
3a <The main program 3a>≡
main('argc, 'argv)
    int argc;
        /* number of arguments on UNIX command line */
    char **argv;
        /* the arguments, an array of strings */
    {
        <Variables local to main 3b>
        prog_name = argv[0];
        <Set up option selection 3c>
        <Process all the files 4a>
        <Print the grand totals if there were multiple files 7a>
        exit(status);
    }
```

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.

```
3b <Variables local to main 3b>≡
    int file_count;
        /* how many files there are */
    char *which;
        /* which counts to print */

3c <Set up option selection 3c>≡
    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.

```
4a <Process all the files 4a>≡
    argc--;
    do {
        <If a file is given, try to open *(++argv); continue if unsuccessful 4d>
        <Initialize pointers and counters 5b>
        <Scan file 6a>
        <Write statistics for file 6c>
        <Close file 4e>
        <Update grand totals 6d>
        /* 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.

```
4b <Variables local to main 3b>+≡
    int 'fd = 0;
        /* file descriptor, initialized to stdin */

4c <Definitions 2c>+≡
    #define READ_ONLY 0
        /* read access code for system open */

4d <If a file is given, try to open *(++argv); continue if unsuccessful 4d>≡
    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;
    }

4e <Close file 4e>≡
    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.

```
4f <Definitions 2c>+≡
    #define buf_size BUFSIZ
        /* stdio.h BUFSIZ chosen for efficiency */
```

```

5a <Variables local to main 3b>+≡
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 */

```

```

5b <Initialize pointers and counters 5b>≡
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?)

```

5c <Global variables 2d>+≡
long tot_word_count, tot_line_count,
    tot_char_count;
    /* total number of words, lines, chars */

```

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.

```
6a <Scan file 6a>≡
while (1) {
    <Fill buffer if it is empty; break at end of file 6b>
    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.

```
6b <Fill buffer if it is empty; break at end of file 6b>≡
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`.

```
6c <Write statistics for file 6c>≡
wc_print(which, char_count, word_count,
         line_count);
if (file_count)
    printf(" %s\n", *argv); /* not stdin */
else
    printf("\n");          /* stdin */
```

```
6d <Update grand totals 6d>≡
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.

```
7a <Print the grand totals if there were multiple files 7a>≡
    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.

```
7b <Definitions 2c>+≡
    #define print_count(n) printf("%8ld", n)

7c <Functions 7c>≡
    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;
                    }
            }
    }
```

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

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. Underlined indices denote definitions; regular indices denote uses.

`stdout`: 2d, 2d