## Software

### Another Approach to Multiple Changefiles

Klaus Guntermann and Wolfgang Rülling*
Technische Hochschule Darmstadt

As reported by W. Appelt and K. Horn in TUGboat Vol. 7, No. 1, pp. 20–21, there are several reasons to allow multiple changefiles in the development of a WEB program. These need not to be repeated here. But we did not follow the same approach when we faced the problem. We decided to develop a separate program which we called TIE, since it ties several parts of a WEB together.

Furthermore we allow that a changefile modifies parts that were just changed. The general strategy is that the addition of changefile $f_{i+1}$ behaves as if the changefiles $f_1$ to $f_i$ had been merged into the WEB program before.

We use a separate program because of the following reasons:

- This single simple program makes additional changes to two programs (namely TANGLE and WEAVE) unnecessary.
- A WEB software developer needs a tool to incorporate frozen changes into a new release of his WEB program from time to time. If the preprocessor program can either create a single changefile or merge all changes into a new WEB file modifications can be written and tested via an additional changefile without touching the released source file. Finally the changes are added ("tied") to the new release.
- TIE can be used for other WEB like systems, too, e.g. for a C version of WEB we created recently. Furthermore TIE allows the application of the changefile method to "plain" Pascal (or even FORTRAN, or whatever programming language you like). One can just merge the changes into the program by selection of the "create new WEB file" option. This is possible since TIE just knows about the line oriented structure of changefiles and has not to deal with the WEB control sequences for sections and so on in the main WEB file. Even data files — as long as they contain textual data — might be changed this way.

The only drawback compared to the method suggested by Appelt and Horn seems to be that it introduces another preprocessing step. This takes some time when the changefiles for large programs like TEX or METAFONT have to be tied since the complete WEB source must be read once more.

Comments to our approach are welcome. TIE is available as a WEB program and can be obtained for a handling charge from Klaus Guntermann at Technische Hochschule Darmstadt.

* now at Universität des Saarlandes, Saarbrücken

### WEB Adapted to C

Klaus Guntermann and Joachim Schrod
Technische Hochschule Darmstadt

In the UNIX environment the programming language C usually is best developed. For systems programming it seems even to be more suitable than Pascal. This led us to the development of CWEB that allows literate programming in C, giving the full documentation tools that WEB adds to Pascal.

The CWEB processors (simply named by a C- prefix, as is the whole system — there was no common relative whose initials the implementors could choose) CTANGLE and CWEAVE are derived from their WEB counterparts.

Changing TANGLE to build a C program instead of a program for the Pascal compiler was rather straightforward. The only problems occurred with the C preprocessor statements that must be allowed in a CWEB program. These statements are supposed to start on a new line, may span several lines ending with a backslash, and in the last line (which may be the first) no other text is allowed to follow. With new tokens designating start and end of a preprocessor statement it was rather easy to add the necessary rules.

Adapting WEAVE to parse C was a heavier task. One of the reasons is that the beginning of a C function declaration cannot be detected very easily since declaration and call of a function look similar if one does not look ahead very far. The look ahead is nearly impossible if CWEB sections are used for the parameter declaration or the function body. We introduced a new (i.e. in addition to WEB) control sequence `@h` that marks the start of a function heading in a declaration.

The grammar had to be rewritten completely. We tried to overcome some of the problems that WEAVE has with Pascal formatting if there is not a bunch of explicit formatting commands. The goal was to get satisfactory results even if explicit formatting is seldom used. (So we are not happy that we had to introduce the control sequence for function headings!)

The CWEBMAC file that specified the layout of the formatted output was redesigned. One major aim was to allow modification of the TEX layout parameters, e.g. tolerance, penalty and paragraph formatting within the documentation parts of the program. (Modification of the tolerance parameter is essential if you try to comment a program in German unless you accept a lot of over-/underfull hbox messages.) In addition one can choose the amount of indentation in the program part.

Following we give a CWEB version of the probably well known UNIX wc command that counts lines, words and characters in a (list of) file(s).

CWEB is currently written in WEB. A rewrite in CWEB is planned but not yet started.

CWEB can be distributed for a handling charge of DM 150.00 (US-$ 60.00) for educational/research sites, DM 250.00 (US-$ 100.00) for others, on magnetic tape (1600 or 6250 bpi, EBCDIC or ASCII, please state format) or DOS 360 K byte floppy disk.

**1. CWEB-Example.** This example presents the "word count" program from UNIX. We rewrote it in CWEB to demonstrate literate programming in C.

**2.** Most CWEB programs share a common structure. Differences are found merely when all functions are just introduced when needed without any special sequence.

```
    ⟨ global #includes 3 ⟩
    ⟨ global variables 4 ⟩
    ⟨ all functions 17 ⟩
    ⟨ main 6 ⟩
```

**3.** We must include the standard I/O definitions to use formatted output to *stdout* and *stderr*.

```
⟨ global #includes 3 ⟩ ≡
#include ⟨stdio.h⟩
```
This code is used in section 2.

**4.** As global variables we introduce some counters that take the cumulated values for each file.

```
⟨ global variables 4 ⟩ ≡
    long wordct, linect, charct;
```
See also section 5.

This code is used in section 2.

**5.** In case that we have to process a list of files we must sum up the grand total in another set of variables.

```
⟨ global variables 4 ⟩ +≡
    long twordct, tlinect, tcharct;
```

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

```
⟨ main 6 ⟩ ≡
main (argc, argv)
    int argc;
    char ∗ ∗ argv;
{
    ⟨ local variables of main 16 ⟩
    ⟨ set up option selection 7 ⟩
    ⟨ go and process all the files 8 ⟩
    ⟨ add the grand total line if there were multiple
            files 15 ⟩
    exit(status);
}
```
This code is used in section 2.

**7.** The first argument should be able to select the counters the user needs. Each selection is given by the initial character (lines, words or characters). We do not process this option string now. It is sufficient just to suppress unwanted figures at output time. However, if no such option was given, print all three values.

```
⟨ set up option selection 7 ⟩ ≡
    wd = "lwc";
    if (argc > 1 ∧ ∗argv[1] ≡ '-') {
        wd = ++argv[1]; argc−−; argv++;
        }
```
This code is used in section 6.

**8.** Now we scan all the arguments and try to open a file, if there is an entry left. The file is processed and its statistics are written. We update the grand total counts anyway.

```
⟨ go and process all the files 8 ⟩ ≡
    i = 1;
    do {
        ⟨ if a file is given we should try to open
                it 9 ⟩
        ⟨ initialize pointers and counters 10 ⟩
        ⟨ scan this file 11 ⟩
        ⟨ write this file's statistics 13 ⟩
        close(f); ⟨ update grand totals 14 ⟩
        }
    while (++i < argc);
```
This code is used in section 6.

**9.** Now we should open the next file. A special trick allows us to handle input from *stdin*—usually file number 0—if no file name was given at all. In this case we use the preset value 0 for *f* to read input from. If we could not open a file, we set the return code for the program an try to proceed for the other parameters.

```
⟨ if a file is given we should try to open it 9 ⟩ ≡
    if (argc > 1 ∧ (f = open(argv[i], 0)) < 0) {
        fprintf(stderr, "wc:␣cannot␣open␣%s\n",
                argv[i]); status = 2; continue;
        }
```

This code is used in section 8.

**10.** Since buffered I/O speeds up things very much we use it, but we do the buffering ourselves. This means that we have to set up pointers and counters such that something is read into the buffer on the first try.

```
⟨ initialize pointers and counters 10 ⟩ ≡
    p1 = p2 = b; linect = 0; wordct = 0;
    charct = 0; token = 0;
```

This code is used in section 8.

**11.** Now we scan the file. The *token* variable indicates if we are just within a word.

```
⟨ scan this file 11 ⟩ ≡
    while (1) {
        ⟨ fill buffer if it is empty 12 ⟩
        c = *p1++;
        if ('␣' < c ∧ c < '177) {
            if (¬token) {
                wordct++; token++;
                }
            continue;
            }
        if (c ≡ '\n') linect++;
        else if (c ≢ '␣' ∧ c ≢ '\t') continue;
        token = 0;
        }
```

This code is used in section 8.

**12.** Using buffered I/O makes it very easy to count the number of characters in the file, almost for free.

```
    define BUFFERSIZE = 512
⟨ fill buffer if it is empty 12 ⟩ ≡
    if (p1 ≥ p2) {
        p1 = b; c = read(f, p1, BUFFERSIZE);
        if (c ≤ 0) break;
        charct += c; p2 = p1 + c;
        }
```

This code is used in section 11.

**13.** Output of the statistics is done in a function. This makes it easy to use it for the totals, too. Additionally we must decide here if we know the name of the file we have processed or if it was just *stdin*.

```
⟨ write this file's statistics 13 ⟩ ≡
    wcp(wd, charct, wordct, linect);
    if (argc > 1) {
        printf("␣%s\n", argv[i]);
        }
    else printf("\n");
```

This code is used in section 8.

**14.** Grand totals are just cumulated.

```
⟨ update grand totals 14 ⟩ ≡
    tlinect += linect; twordct += wordct;
    tcharct += charct;
```

This code is used in section 8.

**15.** Before we stop we have to check if grand total output is needed.

```
⟨ add the grand total line if there were multiple
            files 15 ⟩ ≡
    if (argc > 2) {
        wcp(wd, tcharct, twordct, tlinect);
        printf("␣total\n");
        }
```

This code is used in section 6.

**16.** In this section we declare all the local variables of main. This allows to check how many **register** variables were actually used. The C compiler might ignore some of them if there are too many.

```
⟨ local variables of main 16 ⟩ ≡
    register char *p1, *p2;
    register int c;
    int i, token;
    int status = 0;
    char *wd;
    char b[BUFFERSIZE];
    int f = 0;
```

This code is used in section 6.

**17. Printing the output lines.** According to the given options we print the values. If an invalid option character was given, we do an emergency stop. But the user is informed about legal parameter settings.

We use a CWEB macro for output of the counts. That makes it easy to show all values in identical format. Of course, a C preprocessor macro would have done the job, too. But that wouldn't have shown the CWEB macro feature.

Because the function is rather short we do not split it into subsections.

```
    define prt_value(#) ≡ printf("%7ld", #);
            break;
⟨ all functions 17 ⟩ ≡
void wcp (wd, charct, wordct, linect)
    char *wd;
    long charct, wordct, linect;
{
    register char *wdp = wd;
    while (*wdp) {
        switch (*wdp++) {
            case 'l': prt_value(linect)
            case 'w': prt_value(wordct)
            case 'c': prt_value(charct)
            default: fprintf(stderr,
            "usage:␣wc␣[-clw]␣[name␣...]\n");
                exit(2);
            }
        }
    return ;
}
```

This code is used in section 2.

**18. Index.** The index is set up by CWEAVE.