Listing 4

The Problem ..... §1
The Solution ..... §2
Processing input lines ..... §6
Command line options ..... §17
Error messages ..... §21
References ..... §24
Index ..... §25

1. The Problem. Rather than make up a problem, we'll attempt to solve an exercise from *The C Programming Language* [1] using `CWEB` rather than plain C. In particular, we've chosen the following problem from page 34 of the second edition:*

> **Exercise 1–22.** Write a program to "fold" long input lines into two or more shorter lines after the last non-blank character that occurs before the n-th column of input. Make sure your program does something intelligent with very long lines, and if there are no blanks or tabs before the specified column.

This seems to imply that $n = 80$ requires output lines to contain at most 79 columns. It seems a bit more logical to let the user specify the maximum number of columns in the output, so that $n = 80$ will give us output lines of ≤ 80 characters. Since the problems are isomorphic (a solution to either is an "off by one" error for the other), we choose to solve the latter.

2. The Solution. The structure of our program is fairly standard, and doesn't need a lot of explanation. After processing any command line options, we copy input lines to the standard output, folding them when necessary.

Since we process all the options before we process any files, different options cannot be used for separate files.

```
⟨ Header files 3 ⟩
⟨ Global variables 5 ⟩
⟨ Functions 7 ⟩;
main(int argc, char *argv[])
{
  ⟨ Scan command line options 19 ⟩;
  ⟨ Allocate space for the input buffer 4 ⟩;
  ⟨ Copy the input to the standard output, folding lines as
      necessary 6 ⟩;
  return EXIT_SUCCESS;
}
```

3. ⟨ Header files 3 ⟩ ≡

```
#include <stdlib.h>     /* for EXIT_SUCCESS */
```

See also sections 8 and 14.

This code is used in section 2.

* A similar problem appears on page 31 of the first edition.

4. Since we don't want to place any unnecessary restrictions on line length, or on allowable values of n, we allocate space for the input buffer dynamically. We grab one character more than we need for a line, just in case a line contains exactly *fold_column* characters or we have a space in exactly the perfect spot (plus a byte for the `'\0'`, of course).

Since this is the only memory allocation we do, the *malloc* shouldn't fail, but you never can tell.

```
⟨ Allocate space for the input buffer 4 ⟩ ≡
  buffer ← (char *) malloc(fold_column + 2);
  if (buffer ≡ Λ) {
    ⟨ Announce that we ran out of heap space 22 ⟩;
    exit(EXIT_FAILURE);
  }
```

This code is used in section 2.

5. Unless the user specifies otherwise, we assume that folding will occur after the 80th column.

```
#define DEFAULT_FOLD 80U
⟨ Global variables 5 ⟩ ≡
  char *buffer;
  size_t fold_column ← DEFAULT_FOLD;
```

See also section 18.

This code is used in section 2.

---

6. Processing input lines. We assume that once we're ready to deal with input lines, the contents of *argv* have been "normalized"—that all arguments that do not represent filenames have been replaced with null pointers. This will make our job a bit easier.

We also assume that the string `"-"` used as a filename refers to the standard input.

```
⟨ Copy the input to the standard output, folding lines as
      necessary 6 ⟩ ≡
  if (⟨ No file names were specified 17 ⟩)
    fold_file("-");
  else {
    int i;
    for (i ← 1; i < argc; i++) {
      if (argv[i] ≠ Λ)
        fold_file(argv[i]);
    }
  }
```

This code is used in section 2.

7. The actual line-folding is done by the function *fold_file*, which takes the name of a file to be folded as its only argument. The filename `"-"` is taken to mean "use the standard input."

```
⟨ Functions 7 ⟩ ≡
  void fold_file(const char *filename)
  {
    FILE *infile;
    ⟨ Local variables for fold_file 10 ⟩;
    if (strcmp(filename, "-") ≡ 0)
      infile ← stdin;
    else {
      infile ← fopen(filename, "r");
      if (infile ≡ Λ) {
        ⟨ Warn the user that we couldn't open filename 21 ⟩;
        return;
      }
    }
    ⟨ Copy infile to stdout, folding lines as necessary 9 ⟩;
    if (infile ≠ stdin)
      fclose(infile);
  }
```

This code is used in section 2.

8.
```
⟨ Header files 3 ⟩ +≡
#include <stdio.h>
#include <string.h>
```

9. Whenever we fold an input line, we leave the portion after the fold in *buffer*. We use *left_overs* to let us know how much of *buffer* has already been used, and consequently, how much space is available for reading the rest of the line. The extra 2 characters specified in "*fold_column* − *left_overs* + 2" are for the `'\n'` and `'\0'`.

```
⟨ Copy infile to stdout, folding lines as necessary 9 ⟩ ≡
  while (fgets(buffer + left_overs, fold_column − left_overs + 2,
        infile) ≠ Λ) {
    ⟨ Fold input line in buffer, if necessary 11 ⟩;
  }
  if (left_overs ≠ 0)      /* incomplete last line */
    fprintf(stdout, "%.*s", (int) left_overs, buffer);
```

This code is used in section 7.

10.
```
⟨ Local variables for fold_file 10 ⟩ ≡
  size_t left_overs ← 0;
```

This code is used in section 7.

11. Only lines that overfill the buffer need to be folded.

```
⟨ Fold input line in buffer, if necessary 11 ⟩ ≡
  if (strlen(buffer) ≤ fold_column ∨ buffer[fold_column] ≡ '\n') {
    fputs(buffer, stdout);
    left_overs ← 0;
  } else {
    ⟨ Determine an appropriate folding point, and fold the line 12 ⟩;
  }
```

This code is used in section 9.

12. We use an auxiliary pointer, *ptr*, to find an appropriate place to fold the line.

```
⟨ Determine an appropriate folding point, and fold the line 12 ⟩ ≡
  {
    char *ptr;      /* pointer to the folding point */
    ⟨ Point ptr at the appropriate place in buffer for the fold 13 ⟩;
    ⟨ Fold buffer at the place specified by ptr 16 ⟩;
  }
```

This code is used in section 11.

13. We define "appropriate place" as "just after the leftmost non-blank preceding the rightmost space in the buffer." The first loop finds the rightmost space; the second finds the non-blank.

```
⟨ Point ptr at the appropriate place in buffer for the fold 13 ⟩ ≡
  for (ptr ← buffer + fold_column; ptr > buffer ∧ ¬isspace(*ptr);
       ptr--) ;
  for ( ; ptr > buffer ∧ isspace(*ptr); ptr--) ;
  ++ptr;      /* point after the non-blank */
  ⟨ Deal with specification flaw 15 ⟩;
```

This code is used in section 12.

14.
```
⟨ Header files 3 ⟩ +≡
#include <ctype.h>
```

15. There is a flaw in the original problem specification. We must do something intelligent if there are no blanks in the input line, but what if there are no *non*-blanks? A number of alternatives present themselves, none of them very pretty. We choose to set things up so a line of spaces gets printed, regardless of the consequences should this program be ported to an operating system that insists on removing trailing blanks from text lines.

```
⟨ Deal with specification flaw 15 ⟩ ≡
  if (ptr ≡ buffer + 1 ∧ isspace(buffer[0]))
    ptr ← buffer + fold_column;
```

This code is used in section 13.

16. If *ptr* does not point to a space, it can only mean there was no space in an appropriate column, so we print the maximum characters allowed, and leave the extra character (*buffer*[*fold_column*]) for the next line. Otherwise, we print the characters before the fold, move the characters after the fold to the front of the buffer, and set *left_overs* accordingly.

The problem does not specify what we should do with the blanks that occur "on the fold." We pass them through after the fold, but if we want to remove them, we can add " **while** (*isspace*(*ptr*)) *ptr*++;" before the *strcpy*.

```
⟨ Fold buffer at the place specified by ptr 16 ⟩ ≡
  if (¬isspace(*ptr)) {
    fprintf(stdout, "%.*s\n", (int) fold_column, buffer);
    buffer[0] ← buffer[fold_column];
    left_overs ← 1;
  } else {
    fprintf(stdout, "%.*s\n", (int) (ptr − buffer), buffer);
    strcpy(buffer, ptr);
    left_overs ← strlen(buffer);
  }
```

This code is used in section 12.

---

17. Command line options. In section 6, we need to know if any filenames are specified on the command line. We use *opt_count* to keep track of the number of arguments that are options rather than file names. Thus, *opt_count* equal to *argc* implies there were no file names on the command line.

```
⟨ No file names were specified 17 ⟩ ≡
  (opt_count ≡ argc)
```

This code is used in section 6.

18. We treat *argv*[0] as if it were an option—it certainly isn't the name of an input file—to make the above formula work. We therefore initialize *opt_count* to 1 rather than 0.

```
⟨ Global variables 5 ⟩ +≡
  int opt_count ← 1;
```

19. Any command line argument that begins with `'-'`, except for the string `"-"`, is considered an option. Since we process all the options in a single loop, conflicts are resolved in favor of the option specified later in the argument list.

```
#define is_option(v)  (*(v) ≡ '-' ∧ *(v + 1) ≠ '\0')
⟨ Scan command line options 19 ⟩ ≡
  {
    int i;
    for (i ← 1; i < argc; i++) {
      if (is_option(argv[i])) {
        ⟨ Process the option in argv[i] 20 ⟩;
        ++opt_count;
        argv[i] ← Λ;      /* "normalize" the arg list */
      }
    }
  }
```

This code is used in section 2.

20. The only option we support (at present) is '*-n*', where *n* is a number that specifies the maximum number of columns allowed on an output line.

```
⟨ Process the option in argv[i] 20 ⟩ ≡
  if (isdigit(argv[i][1])) {
    fold_column ← (size_t) strtoul(argv[i] + 1, Λ, 10);
  } else {
    ⟨ Tell user about unknown option in argv[i] 23 ⟩;
  }
```

This code is used in section 19.

---

21. Error messages. In a literate program, it is often helpful to the reader if all of the error handling code is described in the same place. This also helps the programmer make sure that the style of the messages is consistent throughout the program.

```
⟨ Warn the user that we couldn't open filename 21 ⟩ ≡
  fprintf(stderr, "I␣couldn't␣open␣the␣file␣\"%s\"\n.", filename);
```

This code is used in section 7.

22.

```
⟨ Announce that we ran out of heap space 22 ⟩ ≡
  fprintf(stderr,
      "I␣couldn't␣allocate␣needed␣memory.␣␣Sorry.\n");
```

This code is used in section 4.

23.

```
⟨ Tell user about unknown option in argv[i] 23 ⟩ ≡
  fprintf(stderr,
      "I␣don't␣know␣the␣'%s'␣option;␣I'll␣ignore␣it.\n",
      argv[i]);
```

This code is used in section 20.

---

24. References.

[1] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. Prentice-Hall, second edition, 1988.

25. Index.

"-" as a filename: 6, 7, 19.
argc: 2, 6, 17, 19.
argv: 2, 6, 18, 19, 20, 23.
argv normalization: 6, 19.
buffer: 4, 5, 9, 11, 13, 15, 16.
DEFAULT_FOLD: 5.
exit: 4.
EXIT_FAILURE: 4.
EXIT_SUCCESS: 2, 3.
fclose: 7.
fgets: 9.
filename: 7, 21.
fold_column: 4, 5, 9, 11, 13, 15, 16, 20.
fold_file: 6, 7.
fopen: 7.
fprintf: 9, 16, 21, 22, 23.
fputs: 11.
i: 6, 19.
incomplete specifications: 15, 16.
infile: 7, 9.
is_option: 19.
isdigit: 20.
isspace: 13, 15, 16.
left_overs: 9, 10, 11, 16.
main: 2.
malloc: 4.
opt_count: 17, 18, 19.
paranoid error checks: 4.
portability problems: 15.
possible improvements: 16, 20.
ptr: 12, 13, 15, 16.
stderr: 21, 22, 23.
stdin: 7.
stdout: 9, 11, 16.
strcmp: 7.
strcpy: 16.
strlen: 11, 16.
strtoul: 20.

⟨ Allocate space for the input buffer 4 ⟩ Used in section 2.
⟨ Announce that we ran out of heap space 22 ⟩ Used in section 4.
⟨ Copy the input to the standard output, folding lines as necessary 6 ⟩ Used in section 2.
⟨ Copy *infile* to *stdout*, folding lines as necessary 9 ⟩ Used in section 7.
⟨ Deal with specification flaw 15 ⟩ Used in section 13.
⟨ Determine an appropriate folding point, and fold the line 12 ⟩ Used in section 11.
⟨ Fold input line in *buffer*, if necessary 11 ⟩ Used in section 9.
⟨ Fold *buffer* at the place specified by *ptr* 16 ⟩ Used in section 12.
⟨ Functions 7 ⟩ Used in section 2.
⟨ Global variables 5, 18 ⟩ Used in section 2.
⟨ Header files 3, 8, 14 ⟩ Used in section 2.
⟨ Local variables for *fold_file* 10 ⟩ Used in section 7.
⟨ No file names were specified 17 ⟩ Used in section 6.
⟨ Point *ptr* at the appropriate place in *buffer* for the fold 13 ⟩ Used in section 12.
⟨ Process the option in *argv*[*i*] 20 ⟩ Used in section 19.
⟨ Scan command line options 19 ⟩ Used in section 2.
⟨ Tell user about unknown option in *argv*[*i*] 23 ⟩ Used in section 20.
⟨ Warn the user that we couldn't open *filename* 21 ⟩ Used in section 7.