CSCU9V4 Practical 8:
"My head is going to explode!"

Introduction

As programs grow large, it makes sense to organise the source into files that reflect the purpose of the functions that the program contains.

Re-org Time!

It's very easy to code, and code some more, only to find that a source file has grown too large. Consider the program below. It takes unformatted, and hence unreadable, text and re-formats it so that the text appears as expected. Your job: Take the single source, and re-organise it into `.c` and `.h` files that *make sense*.

```
/*********************************************************
 * Comliments to K. N. King                              *
 *********************************************************/

/* Formats a file of text -- Because readability matters! */

#include <string.h>
#include <stdio.h>

#define FILE "testfile.txt"
#define MAX_WORD_LEN 20
#define MAX_LINE_LEN 60


int main(void)
{
  char word[MAX_WORD_LEN+2];
  int word_len;

  clear_line();

  if((freopen(FILE, "r", stdin)) == NULL) {
        fprintf(stderr, "Exit -1\n");
        return -1;
  }

  for (;;) {
    read_word(word, MAX_WORD_LEN+1);
    word_len = strlen(word);
    if (word_len == 0) {
      flush_line();
      return 0;
    }

    if (word_len > MAX_WORD_LEN)
```

```
    word[MAX_WORD_LEN] = '*';
   if (word_len + 1 > space_remaining()) {
    write_line();
    clear_line();
   }
   add_word(word);
  }
}

int read_char(void)
{
  int ch = getchar();

  if (ch == '\n' || ch == '\t')
   return ' ';
  return ch;
}

/***********************************************************
 * read_word: Reads the next word from the input and      *
 *            stores it in word. Makes word empty if no   *
 *            word could be read because of end-of-file.  *
 *            Truncates the word if its length exceeds    *
 *            len.                                        *
 ***********************************************************/

void read_word(char *word, int len)
{
  int ch, pos = 0;

  while ((ch = read_char()) == ' ')
   ;
  while (ch != ' ' && ch != EOF) {
   if (pos < len)
    word[pos++] = ch;
   ch = read_char();
  }
  word[pos] = '\0';
}


/***********************************************************
 * NOTE: These are globals. Though unconventional, it     *
 *       works for us, for now. These should go wherever  *
 *       clear_line() goes.                               *
 ***********************************************************/
char line[MAX_LINE_LEN+1];
int line_len = 0;
int num_words = 0;



/***********************************************************
 * clear_line: Clears the current line.                   *
```

```
 *********************************************************/
void clear_line(void)
{
  line[0] = '\0';
  line_len = 0;
  num_words = 0;
}

/*********************************************************
 * add_word: Adds word to the end of the current line.    *
 *           If this is not the first word on the line,   *
 *           puts one space before word.                  *
 *********************************************************/
void add_word(const char *word)
{
  if (num_words > 0) {
    line[line_len] = ' ';
    line[line_len+1] = '\0';
    line_len++;
  }
  strcat(line, word);
  line_len += strlen(word);
  num_words++;
}

/*********************************************************
 * space_remaining: Returns the number of characters left *
 *                  in the current line.                  *
 *********************************************************/
int space_remaining(void)
{
  return MAX_LINE_LEN - line_len;
}

/*********************************************************
 * write_line: Writes the current line with               *
 *             justification.                             *
 *********************************************************/
void write_line(void)
{
  int extra_spaces, spaces_to_insert, i, j;

  extra_spaces = MAX_LINE_LEN - line_len;
  for (i = 0; i < line_len; i++) {
    if (line[i] != ' ')
      putchar(line[i]);
    else {
      spaces_to_insert = extra_spaces / (num_words - 1);
      for (j = 1; j <= spaces_to_insert + 1; j++)
        putchar(' ');
      extra_spaces -= spaces_to_insert;
      num_words--;
    }
  }
  putchar('\n');
```

```
}

/*********************************************************
 * flush_line: Writes the current line without           *
 *             justification. If the line is empty, does *
 *             nothing.                                  *
 *********************************************************/
void flush_line(void)
{
  if (line_len > 0)
    puts(line);
}
```

This file, along with sample unformatted text, and a working executable, can be found on the module webpage.

Doing this correctly will require care:

- Group functions by their purpose; each set belongs in its own file(s).
- Recall that .h files are what 'link' .c files!
- My solution implements 3 .c files.

Ask yourself the following questions:

1. Where do comments go?
2. Once functions have been re-org'ed, what about declarations and pre-processor directives?

NOTE: This is where 'projects' in an IDE shine. First create a new project. Then open the given source file independently; **do not include** it in the current project (otherwise there will be two `main()` functions).