Now that we understand what the program should do, it's time to think about a design. We'll start by observing that the program can't write the words one by one as they're read. Instead, it will have to store them in a "line buffer" until there are enough to fill a line. After further reflection, we decide that the heart of the program will be a loop that goes something like this:

```
for (;;) {
  read word;
  if (can't read word) {
     write contents of line buffer without justification;
     terminate program;
  }

  if (word doesn't fit in line buffer) {
    write contents of line buffer with justification;
    clear line buffer;
  }
  add word to line buffer;
}
```

Since we'll need functions that deal with words and functions that deal with the line buffer, let's split the program into three source files, putting all functions related to words in one file (`word.c`) and all functions related to the line buffer in another file (`line.c`). A third file (`justify.c`) will contain the `main` function. In addition to these files, we'll need two header files, `word.h` and `line.h`. The `word.h` file will contain prototypes for the functions in `word.c`; `line.h` will play a similar role for `line.c`.

By examining the main loop, we see that the only word-related function that we'll need is a `read_word` function. (If `read_word` can't read a word because it's reached the end of the input file, we'll have it signal the main loop by pretending to read an "empty" word.) Consequently, the `word.h` file is a small one:

word.h

```
#ifndef WORD_H
#define WORD_H

/**********************************************************
 * 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);

#endif
```

Notice how the `WORD_H` macro protects `word.h` from being included more than once. Although `word.h` doesn't really need it, it's good practice to protect all header files in this way.