The `justify` application

Program: Text Formatting

- Assume that a file named `quote.txt` contains the following sample input:

```
   C     is quirky,  flawed,    and  an
enormous   success.      Although accidents of   history
 surely  helped,   it evidently    satisfied   a   need

    for  a   system  implementation    language    efficient
 enough   to  displace         assembly   language,
   yet sufficiently   abstract   and fluent    to describe
  algorithms   and     interactions    in a   wide   variety
of   environments.
                     --      Dennis     M.         Ritchie
```

Program: Text Formatting

- To run the program from a `bash` prompt, we'd enter the command

```
justify <quote.txt
```

- The < symbol informs the operating system that justify will read from the file `quote` instead of accepting input from the keyboard.
- This feature, supported by UNIX, Windows, and other operating systems, is called ***input redirection.***

Program: Text Formatting

- Output of `justify`:

```
C is quirky,  flawed,  and  an  enormous  success.  Although
accidents of history surely helped, it evidently satisfied a
need for a system implementation language  efficient  enough
to displace assembly language, yet sufficiently abstract and
fluent to describe algorithms and  interactions  in  a  wide
variety of environments. -- Dennis M. Ritchie
```

- The output of `justify` will normally appear on the screen, but we can save it in a file by using ***output redirection:***

```
justify <quote.txt >newquote.txt
```

Program: Text Formatting

- `justify` will delete extra spaces and blank lines as well as filling and justifying lines.
  - "Filling" a line means adding words until one more word would cause the line to overflow.
  - "Justifying" a line means adding extra spaces between words so that each line has exactly the same length (60 characters).
- Justification must be done so that the space between words in a line is equal (or nearly equal).
- The last line of the output won't be justified.

Program: Text Formatting

- We assume that no word is longer than 20 characters, including any adjacent punctuation.
- If the program encounters a longer word, it must ignore all characters after the first 20, replacing them with a single asterisk.
- For example, the word

  ```
  antidisestablishmentarianism
  ```

  would be printed as

  ```
  antidisestablishment*
  ```

Program: Text Formatting

- The program can't write 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.

Program: Text Formatting

- The heart of the program will be a loop:

```
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;
}
```

Program: Text Formatting

- The program will be split into three source files:
  - `word.c`: functions related to words
  - `line.c`: functions related to the line buffer
  - `justify.c`: contains the `main` function
- We'll also need two header files:
  - `word.h`: prototypes for the functions in `word.c`
  - `line.h`: prototypes for the functions in `line.c`

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
```

line.h

```
#ifndef LINE_H
#define LINE_H

/**********************************************************
 * clear_line: Clears the current line.                   *
 **********************************************************/
void clear_line(void);

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

```
/**********************************************************
 * space_remaining: Returns the number of characters left *
 *                  in the current line.                  *
 **********************************************************/
int space_remaining(void);

/**********************************************************
 * write_line: Writes the current line with               *
 *             justification.                             *
 **********************************************************/
void write_line(void);

/**********************************************************
 * flush_line: Writes the current line without            *
 *             justification. If the line is empty, does  *
 *             nothing.                                   *
 **********************************************************/
void flush_line(void);

#endif
```

Program: Text Formatting

- The outline of the main loop reveals the need for functions that perform the following operations:
  - Write contents of line buffer without justification
  - Determine how many characters are left in line buffer
  - Write contents of line buffer with justification
  - Clear line buffer
  - Add word to line buffer
- We'll call these functions `flush_line`, `space_remaining`, `write_line`, `clear_line`, and `add_word`.

Program: Text Formatting

- Before we write the `word.c` and `line.c` files, we can use the functions declared in `word.h` and `line.h` to write `justify.c`, the main program.
- Writing this file is mostly a matter of translating the original loop design into C.

justify.c

```
/* Formats a file of text */

#include <string.h>
#include "line.h"
#include "word.h"

#define MAX_WORD_LEN 20

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

Enter this code using a text editor

```
  clear_line();
  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);
  }
}
```

Enter this code using a text editor

Program: Text Formatting

- `main` uses a trick to handle words that exceed 20 characters.
- When it calls `read_word`, `main` tells it to truncate any word that exceeds 21 characters.
- After `read_word` returns, `main` checks whether `word` contains a string that's longer than 20 characters.
- If so, the word must have been at least 21 characters long (before truncation), so `main` replaces its 21st character by an asterisk.

Program: Text Formatting

- The `word.h` header file has a prototype for only one function, `read_word`.
- `read_word` is easier to write if we add a small "helper" function, `read_char`.
- `read_char`'s job is to read a single character and, if it's a new-line character or tab, convert it to a space.
- Having `read_word` call `read_char` instead of `getchar` solves the problem of treating new-line characters and tabs as spaces.

Program: Text Formatting

- `line.c` supplies definitions of the functions declared in `line.h`.
- `line.c` will also need variables to keep track of the state of the line buffer:
  - `line`: characters in the current line
  - `line_len`: number of characters in the current line
  - `num_words`: number of words in the current line

Building a Multiple-File Program

- Building a large program requires the same basic steps as building a small one:
  - Compiling
  - Linking

Building a Multiple-File Program

- Each source file in the program must be compiled separately.
- Header files don't need to be compiled.
- The contents of a header file are automatically compiled whenever a source file that includes it is compiled.
- For each source file, the compiler generates a file containing object code.
- These files—known as ***object files***—have the extension `.o` in UNIX and `.obj` in Windows.

Building a Multiple-File Program

- The linker combines the object files created in the previous step—along with code for library functions—to produce an executable file.
- Among other duties, the linker is responsible for resolving external references left behind by the compiler.
- An external reference occurs when a function in one file calls a function defined in another file or accesses a variable defined in another file.

Building a Multiple-File Program

- Most compilers allow us to build a program in a single step.
- A GCC command that builds `justify`:

```
gcc -o justify justify.c line.c word.c
```

- The three source files are first compiled into object code.
- The object files are then automatically passed to the linker, which combines them into a single file.
- The `-o` option specifies that we want the executable file to be named `justify`.