

Problem Set 4

Due before midnight on Tuesday, February 13, 2007.

1 Assignment

Many word processors allow one to save a document as a plain text file where each paragraph of the original document appears as a single line of text, and the original line breaks are lost. Such files can be difficult for some programs to process because of the very long lines they can contain. Long lines cause two problems for a program that wants to read the file:

1. The line length cannot be determined in advance, so it is impossible to allocate a buffer long enough to hold the entire line.
2. The line may be too long to display as a single line on the user's screen and must be broken into shorter pieces for display.

This assignment has two parts, one that addresses each problem.

The first part is to implement and test a module that implements a *flex char array* abstract data type. A flex char array is a container data structure like an array, but its length can grow as needed. It supports a function `fgetline()` that reads a line of arbitrary unknown length from an input stream and stores it into a growable buffer. It also supports functions for allocating, managing, and accessing the buffer.

The second part is to implement a command `parfill` that performs a paragraph fill on a text file. This command reads each line of the file in turn, breaks it into one or more pieces of a predefined maximum length, and prints out the pieces. In order to achieve a pleasing result, each output line should be as full as possible, and lines should be broken between words (at whitespace). The exception to this rule is that a word can be broken in the middle if it is too long all by itself to fit on a line. Details of how this should be done are given below. Note that `parfill` will behave a little bit like the linux `fmt` command.

2 Flex Char Arrays

A flex char array (`flexbuf` for short) is a data structure that implements the following functions:

- `flexbuf newFlexbuf(void)` creates and returns a new empty `flexbuf`.
- `void freeFlexbuf(flexbuf fb)` frees storage occupied by `fb`.
- `int fgetline(flexbuf fb, FILE* in)` reads a line of input from stream `in` and stores it and the terminating newline character into `fb`, followed by a null character `'\0'`. It grows `fb` as necessary to accommodate the line and terminating null byte. It returns the number of characters stored into `fb` (not including the terminating null byte).

- `const char* toString(flexbuf fb)` returns a pointer to the buffer contents as a read-only string. This allows the calling program to read the contents of the buffer but not modify it. Because the flexbuf is still intact, it can be reused by another call on `fgetline`. Since allocating and freeing storage incurs a considerable cost, it is more efficient in terms of both time and storage to reuse the buffer when possible.
- `char* extract(flexbuf fb)` transfers ownership of the character buffer to the caller. It does this by detaching the character buffer from the `fb` data structure and making `fb` empty, just as it was when first created. An unrestricted pointer to the detached buffer is then returned to the caller, who is responsible for eventually freeing it.
- `int buflen(flexbuf fb)` returns the current length of the buffer. This number is the allocation size of the buffer, not the length of the string currently stored in the buffer (which could be shorter).

The type `flexbuf` is defined in the interface file `flexbuf.h` as follows:

```
typedef struct flexbuf *flexbuf;
```

This declares the new type `flexbuf` to be a pointer to an object of type `struct flexbuf`. It does not define the fields of the structure, so the only thing that a client program can do with the structure is to pass around pointers to it; it can't look inside. This kind of a partially-defined structure is sometimes called an *opaque* structure.

The only functions that will directly access the fields of a `struct flexbuf` are those defined in the implementation file `flexbuf.c`, which also contains the full definition of `struct flexbuf`:

```
struct flexbuf {
    int len;    // allocation length
    char* buf; // storage area
}
```

The function `newFlexbuf()` creates a new *empty* `flexbuf` object, i.e., one with `len==0` and `buf==NULL`, and returns a pointer to it. Note that `extract` also makes its `flexbuf` argument empty.

Let `fb` be an empty `flexbuf`. Before data can be stored into it, a character storage buffer must be allocated, and `fb->buf` must be set to point to it. The initial size of the storage buffer is given by the constant `INITIAL_CHUNK_SIZE`, which should be defined to be 8 bytes. Don't forget to also store `INITIAL_CHUNK_SIZE` into `fb->len`.

The size of a `flexbuf fb` can be changed using the standard function `realloc()`. `realloc()` modifies the size of the existing buffer if possible. If it can't, it creates a new buffer of the desired size and copies the data from the old buffer into it. This takes time $O(n)$, where n is the original buffer size. `realloc()` returns a pointer to the modified or new buffer. The returned pointer will be different from the old buffer pointer in case it was necessary to allocate a new buffer. Programs should always assume that the new buffer will be different and should therefore always replace the old buffer pointer with the returned pointer.

Whenever `flexbuf fb` needs to be expanded, the new size should always be double the old size. Hence, it starts out with 8 bytes, then grows to 16, 32, 64, and so forth. The advantage to the doubling approach as opposed to just increasing the size by a constant amount each time is that total time spent in `realloc()` is then $O(n)$ rather than $O(n^2)$ which would be the case otherwise.

3 Reading Long Lines

There are many ways to read a line into a buffer. The simplest to code is to use `fgetc()` to read from the stream a character at a time. However, it is much more efficient to read a whole block of characters at once.

For this assignment, you should read the stream using the standard function `fgets()`. It takes three parameters: a buffer pointer p , a buffer size n , and a stream. It reads the stream until either a new line character is encountered or $n - 1$ characters have been read. It places the characters read, including the new line character if encountered, into the buffer. It then places a null byte immediately following the last character read. From these rules, it follows that:

- `fgets()` never writes more than n bytes into the buffer, so it cannot overflow a buffer that is at least n bytes long.
- If `fgets()` wrote a null character to the last buffer slot $p[n-1]$, then it read exactly $n - 2$ bytes from the stream. In that case, if $p[n-2]$ contains a new line character, the entire line has been read. Otherwise, only a portion of the line has been read.
- If `fgets()` did not write to buffer slot $p[n-1]$, then fewer than $n - 1$ characters were read, so reading stopped because `fgets()` encountered a new line character or end-of-file. In either case, the entire line has been read. Note that `fgets()` returns `NULL` to indicate end-of-file, but it only returns this when end-of-file is encountered before reading any characters. This means that the only way to tell whether the last line of a file has its terminating new line character or not is to look for it in the buffer.
- To determine whether or not `fgets()` writes a null character to the last buffer slot, store any non-null character there before calling `fgets()` and check afterwards to see if it has changed.

Note that `fgetline()` should only expand the buffer when it is full and the current input line is not yet complete. Thus, if `fgetline()` is called when the `flexbuf` is not empty, it should first call `fgets()` with the current size of the buffer. Only if the line is not complete after this call returns should the buffer be expanded.

4 Paragraph Fill

Paragraph fill means to format a paragraph as a sequence of lines so that each line is as full as possible without exceeding the maximum desired line length `MAX_LINE`, which for this assignment should be fixed at 65. Lines should be broken at word boundaries whenever possible. When a line is broken, any white space to the left and right of the point of the break should be removed. Similarly, any trailing whitespace on each line should be removed. The only whitespace that should *not* be removed is whitespace that occurs between words of an output line, and leading whitespace on the original input line, which should be treated as if it were not whitespace. A consequence of this rule is that if an input line begins with more than `MAX_LINE` blanks, one or more blank lines should appear in the output.

To make this clearer, you should implement the following algorithm:

1. Read in an entire line using `fgetline()`.

2. If the line is at most `MAX_LINE` characters long, remove trailing whitespace, print it out followed by a new line character, and go back to step 1. (Note that a new line character is itself considered to be whitespace, so any new line present in the input line will get removed along with the trailing whitespace.)
3. Otherwise, scan backwards from the $(\text{MAX_LINE} + 1)^{\text{th}}$ character towards the beginning of the line looking for a break point. A break point is a whitespace character immediately preceded by a non-whitespace character. If a break point is found, break the line at that point. If not, break the line after the `MAX_LINE`th character.
4. Print the portion of the line before the break (after removing any trailing white space), and follow it with a new line. Go back to step 2 and continue processing the remainder of the line, beginning with the first non-whitespace character after the break point.

5 Program Specification

You are to write two modules.

`flexbuf` implements a flex char array as described above. The header file `flexbuf.h` should define an opaque interface as described in section 2. The implementation file `flexbuf.c` should implement the required functions and define the `struct flexbuf` structure.

Module `parfill` implements a paragraph fill command with `MAX_LINE` fixed at 65. It should be compiled and linked with `flexbuf` to produce an executable file `parfill`.

`parfill` expects the name of a text file to be passed to it as a command line argument. The main function should check its command line arguments, open the specified file for reading, and if successful, call a function `processLines()` to process the file. `processLines()` should use `fgetline()` to read the file a line at a time into a `flexbuf`. For each line read, it should call `toString()` to obtain a pointer to the buffer containing the line. It should then call a function `formatLine()` to perform a paragraph fill on the line and print it out as described in section 4.

Because `toString` returns a read-only string, you cannot modify the string. In particular, you cannot insert null characters into the string to break it into smaller strings. Rather, you should use the precision field of the string format specifier `s` and the special `*` format character to tell `printf()` how many characters of the string you want to print. For example, to print `k` characters of the buffer starting with the character pointed to by `next`, you would write:

```
printf( "%.*s", k, next );
```

Here, `“.*”` says to use the next list element `k` for the precision field. The letter ‘`s`’ is the string format specifier.

Both `processLine()` and `formatLine()` are functions that are private to the `parfill` module. They should be declared to be `static`, and prototypes for them should be placed at the top of `parfill.c`. The definitions can go anywhere in the file, but it is conventional to place the definitions after the definition for `main()`.

Before the program terminates, it should free *all* storage that has been malloced. This can be tested by use of the `valgrind` command.

I will supply a program called `unittest.c` for testing `flexbuf`. You should compile my program and link it with your `flexbuf.o` file to produce a `unittest` command. I will also supply some test data files and a test script `testflexbuf` which you should run. It will make several calls on `unittest` and put `flexbuf` through its paces. Any discrepancy between your output

and the sample output indicates an error on your part (or an ambiguity in the problem assignment). Your goal should be zero tolerance of discrepancies!

Note that module `flexbuf` implements the function `extract()` and you can expect `unittest` to test it, but I do not want you to use it in implementing `parfill`. Its purpose is to allow a program to do things like reading an entire file into memory without having to copy each line. However, it is much easier to get into trouble when writing code that modifies memory, so you should use `toString()` in those applications such as `parfill` that can be written perfectly well following a read-only discipline.

6 C Concepts and Tools Needed

This program requires use of a number of new C concepts and tools:

- `struct`, structure tags and opaque structures.
- `typedef`.
- Pointers to characters and C-style null-terminated strings.
- `malloc()`, `realloc()`, and `free()`.
- `fgets()`.
- The string format code `'s'` and its precision argument for `printf()`.
- The use of `'*'` in a format to `printf()`.
- `ctype.h` and the function `isspace()`, which tests for whitespace.
- The `valgrind` debugging tool.

I will try to cover the necessary concepts in Thursday's lecture, but feel free to ask me or the TA's about anything that I've missed or that you don't understand.

Good luck!