Line by Line Parsing in C

Normally, we read input from the keyboard and files in individual tokens that are separated by white space.

scanf and fscanf basically read in successive tokens into the appropriate variables. For many situations, when you know how many tokens you are going to read and what each token represents, this suffices.

The problem becomes when pieces of information have a variable number of tokens. For example, imagine a file that stores full names of people, 1 person per line:

George Washington
John Adams
Thomas Jefferson
James Madison
James Monroe
John Quincy Adams
Andrew Jackson
Martin Van Buren
William Henry Harrison
...

Or, departments at UCF:

Art
Child Family & Communication
Civil and Environmental
School of Nursing
Mol Bio-Operations
Statistics
...

fgets function

Typically, in these sorts of files, some types of white space are delimiters for tokens (tabs and newlines, usually) and other white space (spaces) are not.

The standard way to process these files is to read in one line at a time, and then use a "string tokenizer" to parse out the different pieces in a line.

First, let's look at the function that allows us to read in a whole line at once, fgets:

char *fgets(char *restrict *s*, int *n*, FILE *restrict *stream*);

```
The first parameter represent the string into which you want to read in
the line from the file.
```

The second parameter represents the maximum number of characters you want to read in. (If the line is longer, n characters are read, if the line is shorter, then the whole line is read.)

The third parameter is a pointer to the file from which you want to read.

The function ALSO returns a pointer to the beginning memory address of the character array into which the line was read.

Generally speaking, don't mix fscanf with fgets in the same program. If you call fgets and the "cursor" is reading from the middle of a line, it will read from there to the end of the line, sometimes storing absolutely nothing!

strtok function

If there is only one item per line that you want to read in, then fgets will store that value in the designated character array and you can continue with your program.

However, it's quite possible that a line in a file might have more than one piece of information:

George Washington, Virginia, 67
John Adams, Massachusetts, 90
Thomas Jefferson, Virginia, 82

Here each line contains three pieces of information: the president's name, home state and their age when they died.

Typically, we'd want to store this information into three separate variables and not keep it all in one string. One way to "separate out" the pieces of information given the string is to use a string tokenizer function.

The first call to the function sets up the string tokenizer. You must tell the function which string to tokenize and which items serve as delimiters between tokens. (It the example above, the delimiter is the comma.) Here is how you could call the function given that the whole line from the file was stored in a character array called line. More delimiters can be added in the second string.

```
strtok(line, ",");
```

At the end of this call, line will just store a string that is the first token in its original contents. (The C function strtok just writes '\0' character over the comma.)

Accessing the rest of the tokens

To access each successive token, call the string tokenizer function with NULL as the first parameter and use the same delimiters. The function then returns a pointer to the beginning of the next token. This return value must be stored:

char *p;
p = strtok(NULL, ",");

Thus, continue making this call until there are no more tokens in the string tokenizer. Either the number of tokens in the string will be known, OR you can check to see if p is NULL or not, when it is NULL, there are no more tokens left in the string tokenizer and it has to be "reseeded" with a new string to tokenize. Here is an example of reading in department information separated by tabs:

fgets(line, MAXLEN, fin);
size = atoi(line);

```
deptList = (struct dept*)(malloc(size*sizeof(struct dept)));

for (i=0; i<size; i++) {
        fgets(line, MAXLEN, fin);
     strtok(line, "\t");
        strcpy(deptList[i].name, line);

     p = (char*)strtok(NULL, "\t");
        deptList[i].numFaculty = atoi(p);

     p = (char*)strtok(NULL, "\t");
        deptList[i].totalSalary = atoi(p);
     p = (char*)strtok(NULL, "\t"); // Unnecessary
    }
```

Couple notes about strtok

Since the function strtok returns a void pointer, it needs to be cast to a (char*) as is shown on the previous page.

Notice that no malloc is done for p. This is because the space used is really the space allocated for line. All p does is point to various locations in line.

In particular, each subsequence call to strtok basically writes over the next delimiter with the NULL character and returns a pointer to the character right after the previous NULL character.

So in our example with the departments, line initially looks like this:

```
index   0123    456     78901234    56789
line    Art     22      1196518     54387
```

where each token is tab separated. The indexes into the line array are written above. After the first call to strtok line looks as follows:

```
index   0123    456     78901234    56789
line    Art'\0' 22      1196518     54387
```

Now it makes sense that line points to the first token, even though no new memory was used. By changing the delimiter to the NULL character, line just points to one token.

Now, with the second call to the strtok, the contents of memory look like this:

```
index   0123    456     78901234    56789
line    Art'\0' 22'\0'  1196518     54387
```

Basically, the next delimiter after the first changed delimiter is also changed to the NULL character. The pointer returned is simply pointing to the next memory location after the first NULL character (so p is pointing to the 2 in 22.)

And the process continues until all the delimiters are changed to NULL characters in the string.

One other note: The atoi function converts a string into its corresponding integer. Thus, it takes in "123" and returns 123.