CS 352 (Fall 16): System Programming and UNIX

Project #5

Sorted Linked List

due at 9pm, Wed 5 Oct 2016

1 Overview

This project has a single program! In this program, you will implement a sorted linked list. You will read a series of commands from `stdin`, which will insert and delete elements, and do some other simple operations. You will also implement a "print" feature, which will print the current contents of the list (which will be critical for passing testcases - we'll use it to see if you are keeping the list properly).

(Looking for the "Standard Requirements" ?
We've moved it to the end.)

2 Grading

Things are changing about grading, pay attention!

In this project, we will not provide a grading script for you; instead, you must run the tests manually. The reason for this is that we want you to learn how to do your own testing - since in the Real World, you often will not have some other programmer writing a test suite for you.

Thus, in this project, we will provide testcases (and an example executable), but no grading script; you will have to test things yourself. Our current plan is to not even provide the testcases on the next project.

2.1 Doing the Testing Yourself

Look through the grading script from Project 4 (you can open it with pico or any other text editor). While there will be a fair bit which you won't understand, you will see that there are a number of things that we tested for: comparing `stdout` (has to be exact), comparing `stderr` (more flexibility), checking exit status from commands, and also checking to see if `gcc` produced any warnings.

You are welcome to write your own grading script - or to adapt the one from Project 4. But you are responsible for making it work in Project 5!

2.2 Point Distribution

In this project, your code will be graded by a script, with your score determined by how many testcases you pass for each program. After that score is calculated, a number of penalties may be applied.

2.2.1 Possible Penalties

- -10% - Poor style, indentation, variable names
- -10% - No file header, or missing header comments for any function (other than `main()`)
- -10% - Missing checks of returns codes from standard library functions (such as, but not limited to, `malloc()`)
- -20% - Any use of `scanf("%s")` without limiting the number of characters read. (**NOTE:** This includes any call in the `scanf()` family, including `scanf(), fscanf(), sscanf()`.

3 `turnin`

Turn in the following directory structure using the assignment name `cs352_f16_proj05`:

```
linkedList/
    linkedList.c
```

REMEMBER: Submit only this one directory! Do not place it inside another directory!

4 Program - Linked List

In this program, you will implement a linked list of structs. Each struct will have a pointer to the next node (of course), and two data fields. The first data field will be an array of 20 bytes, which will hold a small string; it represents the name of some person; the second will be an integer, which is their favorite number.

(Although we call the first field a "name", it should be a general string; it might contain any character, including underscores, digits, or symbols. The only limitation is that it will not contain any whitespace - and that it will not be empty.)

At all times, you will keep the list sorted by the names; that is, when you insert a new element into the list, you must place it in the correct position (based on the name). Ignore the "favorite number" field when sorting the list.

When your program begins, the list will be empty; as you run through the testcase, you will update the list (printing it out only when the testcase asks for it).

4.1 Input Format

In this program, each line of input is a command. Each command is designated by a string; some commands also have parameters which follow. Commands and parameters are separted by spaces or tabs but **not** newlines; newlines should only be used at the end of a command.

Note that since all parameters are separated by whitespace, it is not possible to insert a name which has a space inside it. So ``Dr. Anson'' is **not** a valid name - but ``Dr_Anson'' is.

The commands are:

- `print`

  Print the current contents of the list (see details below).

- `insert [name] [num]`

  Insert a new linked list element into the list, at the proper position. If the name already existed on the list, then do **not** update the list; instead, print out an error message to `stderr`.

  Remember that the size of the name field is fixed. If the name given is too long to fit into the name field in your linked list, report an error instead of corrupting your memory.

  However, when reading the name field with `sscanf()` make sure to **leave some space for extra characters.** Take a look at testcase 25; if the user passes a too-long name, they should get an error - even if the last few characters of the name could be interpreted as an integer.

- `delete [name]`

  Search the list for the name. If it exists, delete that node from the list (and do not print anything out). If the name does not exist in the list, then report an error to `stderr`.

- `removeHead`

  Delete the first element from the list. If the list is already empty, then print an error message to `stderr`.

4.1.1 Input Format - Other Details

Each line in the input is limited to 80 characters (including the newline at the end). However, in this program (unlike Program 4), you **must confirm this**. This will require you to check, after each call to `fgets()` or `getline()`, whether you found a newline at the end of the string you read. If a line is longer than 80 characters, you must print an error message to `stderr` and terminate the program. (You may assume that every line in each testcase - including the last - has a terminating newline.)

Within the line, there should be a command, and sometimes some parameters. These elements (up to three of them) are separated by whitespace, and

there can be leading whitespace, trailing whitespace, etc. However, all of the elements must be **on the same line**, meaning that you should never find a newline in the middle of a command.

Empty lines are allowed, and should not be reported as an error. However, a line which is not empty, but which is also not a valid line, should be reported as an error. Remember that a line with only spaces and/or tabs is a line that is non-empty!

Finally, for simplicity, we are not requiring you to detect (or handle) to any junk at the end of the line, after the command and its parameters. If there is junk at the end of a line, you must ignore it. (A more advanced program would handle that as an error case.)

4.2 How to Read the Line

To read a line from input, you must go through two stages:

- **Read the line from `stdin`**
  In this stage, call `fgets()` or `getline()` to read an entire line of input. This ought to run to the end of the line, and then have a newline; however, if the line was too long, you will need to report an error and kill the program.

- Once the line has been read, break it into fields. For this, we **strongly recommend** that you use `sscanf()` to read from the buffer.

  See below for a quick introduction to `sscanf()`.

4.2.1 A Trick for Reading Commands

Not all of the commands take parameters; some take zero, one, or two parameters. However, you can read them all with a single call to `sscanf()`. Call it once, with the maximum number of format specifiers that you will need for the longest command. Then you can simply check the return code from `sscanf()` to see if you all of the pieces which the command requires are present.

4.3 Output

None of the commands should print anything to `stdout`, except for `print`, which prints the entire list. Some other commands print to `stderr` on error; you will also sometimes print to `stderr` when an input line is not valid. (For details, see above.)

The output of the `print` command should be first an integer, which is the current length of the list; then a colon. If the list is empty, the newline should follow the colon; otherwise, print out a space-separated list of name/number combinations. Each should be the name, then a slash, then the number (see the examples below).

4.3.1 Output Example

Imagine that your input file looked like this:

```
print
insert Russ 42
print
insert Eric 8192
insert Rose 10
insert Em   35
print
```

In that case, the output should be as follows.

```
0:
1: Russ/42
4: Em/35 Eric/8291 Rose/10 Russ/42
```

4.4 Error Conditions

- Any line of input which is longer than 80 characters (including the newline) should cause your program to print an error and terminate.
- If `malloc()` ever fails, you should print an error and terminate.
- Any line of input which is not empty, but which does not have a command that you recognize from the list above, should cause your program to print an error message. However, **keep going;** do not terminate the program.
- Likewise, some of the commands listed above have situations where they must print an error message. If these happen, print an error message to `stderr`, but keep running.

4.4.1 Non-Errors

Blank lines of input (with nothing but a newline) are **not errrors;** ignore them silently.

5 `sscanf()`

You have been using `scanf()` in the projects so far; that function reads data from `stdin`. `sscanf()` (see the extra 's'?) reads data from a **string,** but works in more or less the same way.

The first parameter to `sscanf()` is the buffer to read from; the next is the format string, followed by the variables that you want `sscanf()` to read

into. (This is similar to how `fprintf()` takes the output stream as the first parameter.)

Here's a simple example of `sscanf()` in action:

```
char *buf = "foo bar 123 baz";

char  word1[20];
char  word2[20];
int   val;
char  word3[20];

sscanf(buf, "%s %s %d %s", word1, word2, &val, word3);
  // sscanf() returns 4
```

As with `scanf()`, `sscanf()` returns the number of format specifiers matched. It will return 0 if it cannot match any of the specifiers; it can even match EOF if it hits the end of the string!

However, `sscanf()` is a little different than `scanf()` in that it doesn't have any way to "consume" the string - so if you call `sscanf()` multiple times on the same string, you will start the parsing over from scratch, at the beginning. So generally, you will only call `sscanf()` once for each string. (This is exactly how you will use it in this program - use `fgets()` or `getline()` to read the initial string; then use `sscanf()` (once per line) to break the line into fields.

And yes, as you may have guessed: there are similar other variants in the `printf()` and `scanf()` families, such as `sprintf()`, which prints to a string, and `fscanf()`, which reads from an arbitrary file!

6 Standard Requirements

- Your C code should adhere to the coding standards for this class: `http://www.cs.arizona.edu/classes/cs352/fall16/DOCS/coding-standards.html`

- Your programs should indicate whether or not they executed without any problems via their **exit status** - that is, the value returned by `main()`. If you return 0, that means "Normal, no problems." Any other value means that an error occurred.

  In this class, we will always use the value 1 to indicate error; however, outside this class, many programs use many different exit status values - to indicate differnt types of errors.

  In `bash`, you can check the exit status of any command (including your programs) by typing `echo $?` immediately after the program runs (don't run **any** commands in-between).

6.1 Special Note: `scanf()` and `%s`

Several programs in this project (and in later projects) will require you to read strings from `stdin` using `scanf()`. **This can be dangerous, if you read more data than your buffer allows** - since it is possible to read right off the end of the array, and overwrite other memory.

To solve this, `scanf()` allows you to limit the number of characters that you read with the `%s` specifier. **You must always use this feature.** Remember that `scanf()` will also write out a null terminator - so this number **must** be less than the size of your buffer, like this:

```
char buf[128];
int rc = scanf("%127s", buf);
```