CS 352 (Fall 16): System Programming and UNIX

Project #6

Tokenizing Strings

due at 9pm, Wed 26 Oct 2016

1 Overview

In this project, you will learn a little more about pointers - we'll be having you parse a string into "tokens" - that is, strings separated by whitespace.

However, the focus of this project is to get better at testing, and also to get a little experience writing Makefiles. So read the spec carefully! There are lots of brand-new requirements - even though the actual C program is not terribly complicated.

2 Grading

In this project, we have **several changes** about grading. Pay attention to each one!

- We will not provide any testcases. Instead, you must write your own - and **you must turn them in!** (It's a good idea to also write a grading script, but don't turn it in.)
- We will use `gcov` to confirm that your testcases cover all of the code in your program.
- We will use `valgrind` to confirm that your program has no memory errors.
- You must provide a Makefile, which compiles your code.

We'll give details for all four points below.

2.1 Grading Scheme

This project has some grade items which apply to the entire project (not to individual testcases), so our grading scheme has gotten more complex:

- 10% of your grade will come from your Makefile (see requirements below).
- 20% of your grade will come from `gcov` coverage (see requirements below).
- 70% of your grade will come from testing your code, comparing it to the example program.

Of course, just as before, we also have a set of penalties which may be applied - these subtract from your overall score:

- -10% - Copying the tokens out of the buffer, instead of modifying the buffer to add null terminators.
- -40% - Any use of a C library function to tokenize the string.
- -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()` family `%s` specifier without limiting the number of characters read.

2.2 Testcase Grading

We have decided to subdivide the score from each testcase:

- You must exactly match `stdout` to get **ANY** credit for a testcase.
- You will lose one-quarter of the credit for the testcase if the exit status or `stderr` do not match the example executable.
- You will lose one-quarter of the credit for the testcase if `valgrind` reports any errors.

As always, you lose half of your testcase points if your code does not compile cleanly (that is, without warnings using `-Wall`).

2.3 Testcase Names

In this Project, we will not provide any testcases; instead, you must write your own. Name each testcase file the way that you have seen in previous projects; each filename must begin with `test_<programName>_` . For instance, for the program `tokenize`, valid testcase names could be:

```
test_tokenize_foo
test_tokenize_01_thisIsMyFirstTest
test_tokenize_doLotsOfInterestingThings
```

Unlike previous projects, put all testcases **INSIDE** the program directory.

2.4 How Testcases Will Be Used

The testcases you write will be used for two purposes. First, we will run your program, using your own testcases, and then use use `gcov` to make sure that your own testcases cover all of your own code. That is, your own testcases must be sufficient to cover your own code.

However, for checking that your program runs correctly (the main "testcase" checking), we will not simply use your own testcases. If we did, you could then give yourself a good grade by writing easy testcases! Instead, the entire class will use the same testcases - and this will be a mix of instructor and student testcases.

To build this set of testcases, we will start with a small set provided by the instructors; we'll focus on corner cases, errors, etc. We will then (mostly) randomly select a set of testcases, chosen from the many different student testcases. So most of your testcases will be written by your fellow students!

3 `Makefile` - Requirements

In this project, you must provide your own Makefile. The Makefile must include a rule which builds your executable (`tokenize`) from your source file (`tokenize.c`). In addition, your Makefile may (but is not required to) include other rules; common rules include `all` and `clean`.

Your rule must:

- Automatically build your executable from your `.c` file when we type `make` in the directory.
- Must automatically re-build the executable if the `.c` file is modified.
- **NOT** rebuild the executable if the `.c` file has not been modified.
- Use `gcc` as the compiler.
- Pass the `-Wall` option (because you always do!).
- Pass the `-g` option (so that we can run your exectuable under `valgrind`).
- Pass the appropriate arguments (see the slides) so that we can use your executable with `gcov`.

4 `gcov` - Requirements

We will run your program with the testcases that you provide, and will use `gcov` to see if your testcases cover your entire program. Of course, it is very hard to cover the code that tests for bad return values from `malloc()` - so you are not required to cover those. However, you must cover **EVERY OTHER LINE.**

The TAs will check the `gcov` output by hand. Remember that 20% of your grade comes from having your program completely covered. Partial scores will be assigned as follows:

- All lines covered (except for `malloc()` error handling) - 20 points.
- 5 or fewer lines not covered - 10 points.
- More than 5 lines not covered - 0 points.

5 `valgrind` - Requirements

On every testcase, your code must run without valgrind errors. To check for errors, check the last line of the valgrind output; a good run will look roughly like this:

```
==4364== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

If one or more errors occur, the output will look something like this:

```
==3944== ERROR SUMMARY: 36 errors from 2 contexts (suppressed: 0 from 0)
```

If valgrind reports errors, you can find details about them earlier in the output. Here is an example of a write past the end of a malloc'd buffer:

```
==4449== Invalid write of size 8
==4449==    at 0x4C313C7: memset (in /usr/lib/valgrind/vgpreload_memcheck-amd64-linux.so
==4449==    by 0x400E8E: insertIntoList (example_linkedList.c:361)
==4449==    by 0x400BDB: main (example_linkedList.c:194)
==4449==  Address 0x51fc060 is 0 bytes after a block of size 32 alloc'd
==4449==    at 0x4C2AB80: malloc (in /usr/lib/valgrind/vgpreload_memcheck-amd64-linux.so
==4449==    by 0x400E59: insertIntoList (example_linkedList.c:355)
==4449==    by 0x400BDB: main (example_linkedList.c:194)
```

6 Program - Tokenize

This program is simple: read lines of text from `stdin`. Break each line into multiple "tokens" - that is, strings separated by whitespace - and save the strings. After you have read the entire file, print out a quick summary of what you have read, and then all of the tokens.

Of course, the details are a little tricky. First, you must support lines of **any** length - and thus, you are **required** to read the lines with `getline()` instead of `fgets()`. Second, when you parse the string into tokens, you are not allowed to duplicate the tokens! Instead, you must **modify the buffer**, adding null terminators as necessary, to turn your one buffer - with a single string inside it - into many strings. Third, in order to store the information about the tokens, you must `malloc()` an array of `char*` - and store the pointer to those tokens into the array.

To store the data for the various lines, you must use a linked list, with each node representing one line. To accomplish this, declare a struct which has (at least) the following fields:

- A `next` pointer
- A count of the number of tokens on this line (could be zero!)
- A pointer to a malloc'd array of `char*`, giving the pointers to the tokens on this line. (If you like, this can be `NULL` on lines where the count of tokens is zero.)

As you read `stdin`, parse each line into a new list node, and add it to the tail of your list; when you hit EOF, print out all of the information in the list.

6.1 Limitation

As you might guess, there's a C standard library function which will do (most) of this work automatically! However, the point of this assignment is to learn more about pointers and strings - so in this assignment, you **may not use C library functions to tokenize the string!**.

(We'll be happy to show you the standard function after the project is due - or, you're welcome to Google for it. Just don't use it!)

6.2 What is a Token?

You should tokenize the input using the same rules as the `%s` format specifier of `sscanf()`:

- Tokens are separated by whitespace (that is, spaces, tabs, and newlines).
- Leading whitespace should be skipped; that is, the first token begins at the first non-whitespace character.
- Trailing whitespace should be skipped.
- If there are multiple whitespace characters in a row, skip all of them; that is, no token may have length zero.

Thus, a line which contains **only** whitespace (including the special case where a line has only a newline) has zero tokens.

6.3 Input Format

There are very few limitations on the input this time; the only limitation (which you don't have to check) is that we will only include printable ASCII characters in the file. So we won't send arbitrary binary data; but the input could be anything that might be in a text file.

There is no limit on the number of lines, on the number of characters per line, the number of tokens per line, or the total number of tokens.

6.4 Output Format

Your program should not print out anything until it has read the entire input, and stored all of the information into the linked list structure. After reading the input, the program must print out a single summary line, like this:

```
Lines=10 Tokens=42
```

After that, you must print out all of the information in the linked list. Each line will have a summary line, like this:

```
Line=3 Tokens: 7
```

Following the summary line, each token will be printed on a single line, with the following format (note that there are exactly two spaces before the word "Line" on this line):

```
  Line=1 Token=2: "asdf"
```

6.5 Example Output

Here is the output from a small testcase file:

```
Lines=3 Tokens=5
Line=0: Tokens 3
  Line=0 Token 0: "foo"
  Line=0 Token 1: "+="
  Line=0 Token 2: "bar"
Line=1: Tokens 0
Line=2: Tokens 2
  Line=2 Token=0: "%"
  Line=2 Token=1: "z;"
```

This output corresponds to the following input (note the blank line in the middle):

```
foo += bar

% z;
```

Interestingly, it also corresponds to this input (because whitespace is ignored, except for marking the breaks between tokens):

```
            foo         += bar

%                          z;
```

6.6 Error Conditions

There are no error conditions for this program. (Other than, as always, handling bad return values from `malloc()` and such.)

7 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).

8 `turnin`

Turn in the following directory structure using the assignment name `cs352_f16_proj06`:

```
tokenize/
    Makefile
    tokenize.c
    test_tokenize_*     (turn in several of these!)
```

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

9 Grade Preview

48 hours before the project is due, we will run the automated grading script on whatever code has been turned in at that time; we'll email you the result. This will give you a chance to see if there are any issues that you've overlooked so far, which will cost you points.

Of course, this grading script will not include the full set of student testcases, which we will use in the grading at the end, but it will give you a reasonable idea of what sort of score you might earn later on.

We will only do this once for this project. If you want to take advantage of this opportunity, then make sure that you have working code (which compiles!) turned in by 48 hours before the due date.