CS 214: Systems Programming, Fall 2012
Programming Assignment 1: Tokenizer

1 Introduction

In this assignment, you will practice programming with C pointers. Much of the pointer manipulation will come in the form of operating on C strings, although you will be dealing with some pointers to structs as well.

Your task is to write a type and a set of functionsin essence, the equivalent of a Java classthat implements a tokenizer. The tokenizer should accept two strings, the first of which will contain a set of *separator characters* while the second will contain a set of *terms* separated by one or more separator characters. The tokenizer should return the terms in the second string one at a time each term is called a token, hence your program is called a tokenizer. For example, when given the following two strings:

“ ”, “today is a beautiful day”

your tokenizer should return:

“today”, “is”, “a”, “beautiful”, and “day”

When given the following two strings:

“/?”, “/usr/local/?/bin/? share”

your tokenizer should return:

“usr”, “local”, “bin”, and “ share”

A string is a sequence of characters delimeted by double quotes ("). Strings can contain newline or double-quote characters, but special systax is required to contain them and certain other characters. These special characters are represented with escape sequences:

newline \n
horizontal tab \t
vertical tab \v
backspace \b
carriage return \r
form feed \f
audible alert \a
backslash \\\\

question mark \?
single quote \′
double quote \”
octal number \000
hex number \$x$hh

2 Implementation

Your implementation needs to export the interface given in the attached `tokenizer.c` file. In particular, you need to define the type needed to represent a tokenizer and three functions for creating and destroying tokenizer objects and getting the next token. Note that we have only defined the minimal interface needed for external code (e.g., our testing code) to use your tokenizer. You will likely need to design and implement additional types and functions.

A token is a sequence of any ASCII character that does not contain a separator character. Separator characters are provided as a string of one or more ASCII characters. Each pair of tokens are separated by one or more separator characters. Multiple separators may be next to each other (see second example above), and/or at the beginning and/or end of the term string. When this happens, your tokenizer should discard *all* separators.

Your implementation must *not* modify the two original strings in any way. Further, your implementation must return each token as a C string in a character array of the exact right length. For example, the token usr should be returned in a character array with 4 elements (the last holds the character ’\0’ to signify the end of a C string).

You may use string functions from the standard C library accessible through string.h (e.g, `strlen()`). However, you may not use strtok(), strsep() or any similar function that already performs the complete tokenization process.

You should also implement a `main()` function that takes 2 string arguments, as defined above. Each character in the first string is a separator. The second string contains zero or more tokens separated by separator characters. Your `main()` function should print out all the tokens in the second string in left-to-right order. Each token should be printed on a separate line. Here is an example invocation of the tokenizer and its output.

tokenizer ” ” ”today is sunny”

today

is

sunny

Keep in mind that *coding style will affect your grade.* Your code should be well-organized, well-commented, and designed in a modular fashion. In particular, you should design reusable functions and structures, and minimize code duplication. *You should always check for errors.* For example, you should always check that your program was invoked with the minimal number of arguments needed.

Your code should compile correctly (no warnings and errors) with the `-Wall` and either the `-g` or `-O` flags. For example

```
$ gcc -Wall -g -o tokenizer tokenizer.c
```

should compile your code to a debug-able executable named tokenizer without producing any warnings or error messages. (Note that -O and -o are different flags.)

Your code should also be efficient in both space and time. When there are tradeoffs to be made, you need to explain what you chose to do and why.

```
IMPORTANT NOTE: You may write your code on any machine and operating system you desire,
but the code you turn in MUST tar (see below), compile and execute on the iLab machines
or a zero grade will be given.  Be sure to compile and execute your code on an Ilab
machine before handing it in.
```

3 What to turn in

A tarred gzipped file named pa1.tgz that contains a directory called pa1 with the following files in it:

- A `tokenizer.c` file containing all of your code.
- A file called `testcases.txt` that contains a thorough set of test cases for your code, including inputs and expected outputs.
- A `readme.pdf` file that contains a brief description of the program and any great features you want us to notice.

Suppose that you have a directory called `pa1` in your account (on the iLab machine(s)), containing the above required files. Here is how you create the required tar file. (The `ls` commands are just to help show you where you should be in relation to `pa1`. The only necessary command is the tar command.)

$ ls pa1 $ ls pa1 Makefile readme.pdf testcases.txt tokenizer.c $ tar cfz pa1.tgz pa1

You can check your `pa1.tgz` by either untarring it or running `tar tfz pa1.tgz` (see `man tar`).

Your grade will be based on:

- Correctness (how well your code works).
- Quality of your design (did you use reasonable algorithms).
- Quality of your code (how well written your code is, including modularity and comments).
- Efficiency (of your implementation).
- Testing thoroughness (quality of your test cases).