CS 214: Systems Programming
Assignment 0: A Tour of C

0. Abstract

C is somewhat different than many of the languages you may have used in the past. As a way of familiarizing yourself with some of these more direct differences, you'll write a fairly simple program to parse and sort a single string of values. While parsing is not in and of itself a difficult task, writing a parser requires that you deal directly with a few fundamental features of C, namely memory management, use of built-in functions and statewise computational rather than procedural library thinking.

1. Introduction

A parser is a program that classifies or identifies segments of a stream input in to one of several different classes, for instance taking a sentence as a single input, a parser might find each token (word), determine the identity of each token and perform some action if they are given in a certain order, like a terminal program. Your tokenizer will be simpler. All it needs to do is read in a single string, break it in to tokens, classify the tokens in to various types, sort them and output them one per line in order.

You will not know the length of the input string or its constituent tokens beforehand, or how many there are, so you will not know how much memory to allocate or how many pointers you'll need to handle the input string. These are two fundamental issues you must deal with by dynamically allocating memory to hold tokens as you discover them. Manipulation of dynamic memory and the pointer construct are two defining characteristics of the C language. While they grant you additional capabilities, they require additional attention and discipline to use effectively.

2. Methodology

Unlike the rest of your projects, this one will be ***individual***, so that you all have time to find partners and have a decent introduction to C, so that you are more useful to your project partners later.

The input string will contain any number of tokens:

"thing stuff otherstuff blarp"

You must extract and sort these tokens. The tokens will be separated by non-alphabetic characters. You must consider *any* non-alphabetic character a separator. For instance:

"thing1stuff3otherstuff,blarp"

.. has the same tokens as the example input string above. Identifying tokens is a simple enough process. If you start at the front of the string and inspect each character from front to back, the first non-alphabetic character marks the end of the first token, and the first alphabetic character afterward marks the beginning of the next token.

You have three major segments to think about:

A. How to inspect an input string of an unknown length without falling off of the end
B. How to save an unknown number of tokens of unknown length
C. How to sort an unknown number tokens of an unknown length

2.1 Unknown Token Lengths

You can only ask for a given amount of memory to store some data in C. Since you do not know the lengths of the tokens when you start, you'll need to discover them as you go so that you can request enough memory to store them in with malloc().

Any time you do not know exactly how many steps something will take, but know when are you done, is often an excellent time for a loop. You should use pseudocode and rough things out first.

```
while( haven't fallen off the input string )
{
  .. read from input string
  while( haven't found a separator and haven't fallen off the input string )
  {
    .. read from input string
    .. advance to next character
  }
  // either fell off of input string, or found a separator
  if( found a separator )
  {
    .. copy the component string out of the input string
  }
```

You will need to move along the input string testing characters until you find a separator. Remember, any non-alphabetic character is a separator (can't just split()).

Once you find a separator, you can determine the length of the string it marked the end of. You can then request some memory to store that string with malloc() and copy the string over with memcpy(). You now need to hold the pointers to your component string copies in a data structure.

2.2 Unknown Token Numbers

Since you do not know how many total tokens will be in a single input string, you need to be ready with an extensible data structure. There are several ways to approach this problem. You could build a linked list, or manage an array to increase its size as needed, or implement a red/black tree (no hashtables, however). Whichever one you use, the data structure must stretch to hold any number of pointers to tokens as you find them, one after the other, during runtime.

2.3 Token Sorting

Since you must discover the end of a token before you copy it out of the input string and into your list of tokens, you must insert them in to your list one by one. You might find it easier to inspect your list on each insert so that each token is inserted in sorted order, so that your list itself is always in sorted order, or to just insert the tokens as you find them and sort them all at once when you've found them all. When writing your sorting function, remember that it will not implicitly know the lengths of the tokens it is comparing. Be careful not to fall off the ends of your tokens when comparing them.

3. Results

You may use any functions or libraries available in the iLab machines. You will likely want to investigate the functions defined in the string.h library. 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 and degenerate cases. For example, you should always check that your program was invoked with the correct number of arguments. Your code should compile correctly (no warnings and errors) with:

"gcc -O -g -o pointersorter pointersorter.c"

This command 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).

Do not use:

Makefiles - even if you know how
C99 or other versions or subversions of C
fsanitize

Be sure to document your code in a "readme.txt", describing how it operates, and detail your testing in a "testcases.txt". A portion of your grade is allocated to how completely you tested your code. You should also expect, and test for, unfriendly input. For instance, running your code with too few arguments, with too many arguments, with blank strings, with two separator characters together, etc. Your code should handle these situations gracefully. If you decide you must exit, exit nicely with an informative error message. Be sure to free any and all dynamic memory before returning or exiting. Segmentation Fault is never a part of normal operation. Any code that causes a Segmentation Fault will be considered inoperable, regardless of any output and will be heavily penalized.

4. Submission Requirements

Your submission MUST (un)tar (see below), compile and execute on the iLab machines or a zero grade will be given. Be sure to test your code on an iLab machine before handing it in. If it works on your laptop, but does not compile on the iLab machines, it will still net a zero.

Turn in a tarred, gzipped file named "Asst0.tgz" that contains a directory called Asst0 with the following files in it:

A pointersorter.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.txt file that contains a brief description of the program and any great features you want us to notice.

4.1 Creating the tar file

First create a directory named "Asst0" and copy your three files in to it.
Run "tar cfz Asst0.tgz Asst0"
This should create a new file named "Asst0.tgz" that contains your directory and your code.

4.2 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).
Testing thoroughness (quality of your test cases).

4.3 Examples

./pointersorter "thing stuff otherstuff blarp"
thing
stuff
otherstuff
blarp

./pointersorter "thing1stuff3otherstuff,blarp"
thing
stuff
otherstuff
blarp

5. Additional

Be careful to output your strings in descending alphabetical order:

zounds
zoo
marmosets
march
energetically

Treat capital letters as 'greater than' lower case – they should come before lowercase:

for words: aand, aAnd, Aand, Aand
output as:
AAnd
Aand
aAnd
aand

Be robust to error!

expect the wrong number of inputs (return and print out a usage string)
expect only non-letter inputs (print out nothing)
expect empty inputs (print out nothing)
expect terrible, terrible things