Tar Heel Sort - `thsort`

Abstract

In this lab you will implement `thsort`, a program that reads lines from standard input and prints them back in sorted order.

The `sort` program is a classic standard utility. Its purpose is to read lines from standard input (`stdin`), sort the lines, and prints them back out to standard output (`stdout`). It offers many different options for the *type* of sort it performs (lexicographic, numeric, etc). Your implementation will sort in a naive *ascending*, ASCII order, for now.

In Chapter 5, the C book implements a complete, working version of `sort` using *static memory.* Unlike the book's implementation, `thsort` will make use of *heap memory* instead. Your `thsort` program must read arbitrarily long single line of input as arbitrary numbers of lines. To do so, the heap memory your `thsort` allocates will expand dynamically at runtime in order to accomodate the input data.

The purpose of this lab is to practice working with pointers, dynamic memory allocation, and applying knowledge acquired from library documentation.

Setup

Starter Repository

Accept the following GitHub classroom assignment: https://classroom.github.com/a/aEMZiVpi

With your project repository open, click the green "Clone or Download" button. Copy the HTTPS URL. Back in a learncli container, clone this url with the git clone subcommand. Replace with your copied URL. (Right click to paste in Windows.) After cloning, change your working directory to the cloned repository.

Demo

Before you begin, you should try the standard `sort` program:

```
learncli$ sort
the
quick
brown
fox
<CTRL-D>
brown
fox
quick
the
```

The primary use case for `sort` is *not* for interactive use, but rather to sort data piped in from another program or redirected from a file source. In the `data/` directory of your repository you will find a number of files for testing. In the shell, placing a `<` symbol before a filename, after a program, is a special syntax for *redirecting* standard input (`stdin`) to be read from a *file* rather than interactively from the keyboard. Try it out:

```
learncli$ cat data/lines-1.txt
j
c
b
d
h
g
e
a
f
i
learncli$ sort <data/lines-1.txt
a
b
c
d
e
f
g
h
i
j
```

Getting Started

Create a C file named `thsort.c` in the root directory of the repository. For now, get a working `main` function that prints “hello, world” and add the class header:

```
// PID: 123456789
// I pledge the COMP211 honor code.
```

Compiling with `make`

You’ll notice a file named `Makefile` in the project directory if you run `ls`. We will discuss `make` in depth soon and the syntax of a `Makefile` later in the semester. For now, know the `Makefile` contains information on how to build your project. If you run the command `make` in your project directory, the `Makefile` is read and the `all` rule is run. Try it:

```
learncli$ make
gcc -Wall -Wextra -g -std=c11 thsort.c -o thsort
learncli$ ./thsort
hello, world
```

Notice `make` emitted the command `gcc -Wall -Wextra -g -std=c11 thsort.c -o thsort` without you needing to remember and type those options. Later in the course our programs will involve multiple files we need `gcc` to build and then link together. Using `make` and a `Makefile` to automate the specific steps to build a project so that we only need to give the command `make` to rebuild it is a real productivity boost!

Requirements

Your program must be able to read arbitrarily long lines of input, as well as an arbitrary number of lines of input, and sort those lines according to `strcmp`’s lexocographic ordering of string data.

Your program must store its data *on the heap* and specifically *not* in static memory as demonstrated in the book. Your program must not leak any memory and must not perform any invalid heap accesses of memory outside its lifetime. Your program must print “out of memory” to `stderr` and exit immediately if *any* memory allocation call fails.

Your first concern should be producing a working program free of memory leaks. Do this in the simplest, least clever way you can imagine first. Only after you are passing all tests except for the “design challenge” tests, consider how you could redesign your memory allocations to strike a reasonable balance between minimizing the number of reallocations you perform (expensive in *time*) and the amount of memory you allocate (expensive in *space*).

Permitted Library Functions

You are encouraged to read the documentation of each of the library functions below before beginning. You may not need to use all of these functions, but it's likely you would find it beneficial to use most of them. For this lab you are only allowed to use functions from the following list of functions. Any use of a library function outside of these will result in a 20% penalty subtracted from your autograding score.

The names of the functions below are links to their official documentation.

- stdio.h
  - printf - print to standard output
  - fprintf - print to a specific file descriptor (`stderr` for error messages)
  - getchar - for reading input one byte at a time
  - fgets - for reading input blocks of bytes at a time
- stdlib.h
  - malloc - memory allocation
  - calloc - contiguous allocation with 0-initialization
  - realloc - reallocate and resize a dynamic memory block
  - free - freeing dynamically allocated memory
  - qsort - for sorting with the built-in quicksort algorithm
  - exit - for exiting the program prematurely
- string.h
  - strchr - find a character in a string
  - strcmp - compare two strings lexicographically
  - strcpy - copy a string and return a pointer to the end of the result

Helpful Hints

Do not begin with code. On paper, diagram what you will do when you read a line of input. Start from an exceedingly small initial allocation (think: 2 bytes) and diagram each subsequent step you'll need to take to read a line that's longer than your initial allocation.

Do not attempt to write this program in its entirety in your first take; doing so will result in the worst debugging nightmare of your programming career. Cherry pick the a path of test cases that begin with small, simple examples you can pass without much effort and then slowly grow the difficulty of the test cases your program can pass. For some inspiration, peruse the files in the `data` folder and convince yourself some are much less complex to correctly handle than others. Work your way up in difficulty gradually, not all at once.

When you are working at this level of granularity in memory the challenge is *entirely* in the details. You must think very carefully and slowly through where your pointers are pointing, what happens when you reallocate, and how to compute the correct offsets to store additional character data into.

Testing for Memory Leaks

A rule in the `Makefile` is setup to recompile your program, if you've made changes, and then run `valgrind` with to test for memory leaks. Do this as you're working on reading single lines of dynamic length and then many lines. Enter some test data and then press `Ctrl-D` to signal end-of-input. You should see no errors and that *All heap blocks were freed – no leaks are possible.* If you see errors, then go back and carefully consider your memory allocations and what might cause you to be reading or writing *outside of the memory you allocated.* If you see a leak, then go back and be sure anywhere you are allocating memory you are later freeing it.

Once you are nearing the end of the project, you will probably want to test for leaks the larger datasets and not have to type the input in yourself. You can run `valgrind` with input redirected to your program, as well.

```
# Small Test
valgrind --leak-check=full ./thsort <data/lines-256.txt
# Stress Test
valgrind --leak-check=full ./thsort <data/lines-shakespeare.txt
```

Testing Out of Memory Error Handling

Once you are confident your program is not leaking memory and you are passing all tests, you should test how your program behaves when it runs out of memory and your allocations or reallocations fail. To do so, we can self-impose resource limitations on the processes we run in a shell session with the `ulimit` command. Using `ulimit` with the `-v` flag allows you to specify the maximum amount of *virtual memory* a child process can make use of in kilobytes. All of a process' memory is stored in *virtual memory*, so this size must be large enough for the stack, static, program instructions, and shared libraries, in addition to heap data. As such, a value of `8192` kilobytes *should* be large enough to load your program and complete small test cases and small enough to run out of memory for a larger input. For example:

```
learncli$ ulimit -v 8194
learncli$ ./thsort <data/lines-64k.txt
out of memory
```

Note that you can *only decrease this size for a given session.* **To reset this limit, start a new session in your container.** If you accidentally set it too low, you will see errors encountered while loading your program, before it is even able to request heap memory:

```
learncli$ ulimit -v 512
learncli$ ./thsort
Segmentation fault
```