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.

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/k5Zt17zm

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.

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

Suggested Workflow

Step 0. Understanding `fgets` and Buffer Sizes.

Let's start by getting familiar with the `fgets` function and how buffer sizes affect input handling in C.

The following program prompts the user to enter a string. It then stores the input in a fixed-size array on the stack and prints it back. The array where you store the input is called a buffer. Notice that the input string must not exceed 127 characters (plus one for the null terminator), as the buffer size is limited to 128 bytes.

Copy this program into `thsort.c`, compile it, and run it to observe how it behaves. Remember to use `make` to compile the program.

```c
#include <stdio.h>

#define LINE_BUFFER_SIZE 128

int main() {
    char buffer[LINE_BUFFER_SIZE];
    printf("Enter a string: ");
    if (fgets(buffer, sizeof(buffer), stdin) != NULL) {
        printf("You entered: %s", buffer);
    } else {
        printf("No input or error occurred.\n");
    }
    return 0;
}
```

- `fgets` reads a line of input from a given stream (`stdin` in this case) and stores it in the buffer you provide.
- The second argument (`sizeof(buffer)`) tells `fgets` the maximum number of characters to read, including the null terminator (`\0`).
- If the input fits in the buffer, it's stored as a null-terminated string.
- If the input is longer than the buffer, only part of it is read, and the rest remains in the input stream to be consumed later.
- `fgets` returns the same pointer you passed in (`buffer`) on success, and `NULL` on failure (e.g., if end-of-file is reached or an error occurs).

Try This Reduce the buffer size by changing `LINE_BUFFER_SIZE` to something small, like 10. Recompile and run the program again. Enter a long string — what do you notice?

Before continuing, reset the buffer size to 128.

Note on Arrays and Pointers When you pass an array (like `buffer`) to a function, it decays into a pointer to its first element. So in this case:

```
fgets(buffer, sizeof(buffer), stdin);
```

`buffer` is treated as a `char *`, pointing to the start of the array. This is why `fgets` can write into the memory you allocated with `char buffer[LINE_BUFFER_SIZE];`.

Step 1. Storing the buffer on the heap.

Now, modify the program such that it stores the buffer on the heap instead of the stack. Remember to check to make sure that `malloc` was successful.

Once you've made the change, test your modified program using various inputs. Make sure it still reads and prints the string correctly.

Common issue: If you're running into problems, double-check that the size you're passing to `fgets` matches the size of the allocated buffer. Refer back to slide 14 from 3.19 for a refresher on `sizeof`.

Before moving on, make sure your program doesn't have any memory leaks by running it through Valgrind:

```
make leak-check
```

Look at the heap summary at the end of the output. It should include the line:

```
All heap blocks were freed -- no leaks are possible
```

If you see any memory still in use or not freed, double-check that you're freeing everything you've allocated with `malloc` or `realloc`.

Step 2. Reading an infinite number of lines

So far, the program reads a single line, prints it, and then exits. In this step, you'll modify the program so that it continuously reads and prints lines of input until it reaches `EOF` (end-of-file).

To do this, you'll write a function called `readline` that:

- Allocates memory on the heap for each line
- Uses `fgets` to read one line at a time from stdin
- Returns a pointer to the line read or returns `NULL` when there's nothing left to read

Here's the updated main function you should use:

```c
int main() {
    char* buffer;
    printf("Enter a string: ");
    while ((buffer = readline()) != NULL) {
        printf("You entered: %s", buffer);
        printf("Enter a string: ");
        free(buffer);
    }
    return 0;
}
```

Before moving on, make sure test your program's functionality.

Once your program works as expected, ensure that it doesn't have any memory leaks by running it through Valgrind:

```
make leak-check
```

Look at the heap summary at the end of the output. It should include the line:

```
All heap blocks were freed -- no leaks are possible
```

If you see any memory still in use or not freed, double-check that you're freeing everything you've allocated with `malloc` or `realloc`.

Step 3. Storing and Printing All Lines After EOF

In Step 2, you wrote a program that continuously read and printed each line of input until reaching `EOF` (end-of-file). Now, you'll modify the program so that it reads all the lines first, stores them, and then prints them all at once after EOF is reached.

To do this, you'll need an array of pointers, where each pointer will refer to one of the lines read from input.

For now, you can assume the user will enter at most 5 lines. (In a future step, we'll make this dynamic so the program can handle any number of lines.)

Here is the suggested structure.

```
#include <stdio.h>
#include <stdlib.h>

#define LINE_BUFFER_SIZE 128
#define NUM_LINES_BUFFER_SIZE 5

char** readlines(int* num_lines)
char* readline();
void printlines(char** lines, int num_lines);
void freelines(char** lines, int num_lines);

int main() {
    int num_lines = 0;
    char** lines = readlines(&num_lines);
    printlines(lines, num_lines);
    freelines(lines, num_lines);
    return 0;
}
```

Before moving on, make sure test your program's functionality.

Once your program works as expected, ensure that it doesn't have any memory leaks by running it through Valgrind:

```
make leak-check
```

Step 4. Sorting the Strings

In this step, you'll sort the input strings that you stored in the previous step. To do this, you'll use the standard library function `qsort`.

`qsort` `qsort` is a general-purpose sorting function provided by C's standard library (`stdlib.h`). It can sort arrays of any type. You provide it with:

- A pointer to the array
- The number of elements in the array
- The size of each element
- A comparison function that tells `qsort` how to compare two elements.

Here's the function signature:

```
void qsort(void *base, size_t nitems, size_t size, int (*compar)(const void *, const void *));
```

Writing a Comparison Function Since we're sorting strings, we'll compare them using `strcmp`, which returns:

- A negative value if the first string is "less than" the second
- Zero if they're equal
- A positive value if the first is "greater than" the second

Here is the implementation of the comparison function:

```
int cmp(const void *a, const void *b) {
    const char *str1 = *(const char **)a;
    const char *str2 = *(const char **)b;
    return strcmp(str1, str2);
}
```

Note that we cast to `char **` because `qsort` passes in pointers to elements—in this case, pointers to `char *`.

The `const` keyword tells the compiler that a variable's value should not be changed after it's been set. It adds a layer of safety by preventing accidental modification.

Before moving on, make sure test your program's functionality.

Once your program works as expected, ensure that it doesn't have any memory leaks by running it through Valgrind:

```
make leak-check
```

Step 5. Reading an Arbitrary Number of Lines

Up to this point, your implementation can only handle a fixed number of lines. In this step, you'll modify your program so it can handle any number of input lines.

You'll start by allocating space for a limited number of lines. As the user enters more lines than you've allocated for, you'll use `realloc` to resize the array dynamically and make room for more.

How you grow the array is up to you—you might double its size each time, increase it by 50%, or use another strategy.

Before moving on, make sure test your program's functionality.

Once your program works as expected, ensure that it doesn't have any memory leaks by running it through Valgrind:

```
make leak-check
```

Step 6. Reading Lines of Arbitrary Length

In this final part of the lab, you'll extend your program to handle input lines of any length, not just ones that fit within a fixed buffer size.

Just like in Part 5, you'll start by allocating space for a line of reasonable length. If the user types a line that exceeds this size, you'll use `realloc` to dynamically resize the buffer as needed.

Your goal is to keep reading characters until you reach the end of the line (`\n` or `EOF`), expanding the buffer as necessary along the way.

Make sure your program doesn't have any memory leaks by running it through Valgrind:

```
make leak-check
```

Look at the heap summary at the end of the output. It should include the line:

```
All heap blocks were freed -- no leaks are possible
```

If you see any memory still in use or not freed, double-check that you're freeing everything you've allocated with `malloc` or `realloc`.