CS 352 (Fall 17): System Programming and UNIX

Project #7

Linked Lists

due at 11:30pm, Thursday 22 Jul 2021

1 Overview

In this project, you will practice with creating linked lists. You will actually write (nearly) the same program twice - once sorting integers, and once sorting strings. Write the integer version first, as it's a little simpler; in the string version, you will learn about reading arbitrary-length strings from input, and saving them into `malloc()`d buffers.

You should have already used linked lists in previous classes, so I'm not expecting that that part of the project will be new to you. (If it is, come to Office Hours and we'll go through the basics, or post on Piazza) However, I think that this is a good way to practice again with pointers - this time, using pointers between various `structs`.

WARNING: The method for sorting that we use in this project is **famously terrible**. It has $O(n^2)$ performance - while the best sorts have $O(n \log n)$ performance. We're doing it this way because it's an easy way to get practice with inserting into a linked list - but **do not** use this algorithm in a real world program! (When you take 345, you'll learn about better sorts - or, you can read up on QuickSort or Merge Sort on your own time.)

1.1 `structs` as Objects

As you write the code, try to start thinking of a `struct` the same way you would an object in Java or Python - because they really **are** the same thing! To be more precise, an object in Java or Python (or any other language, such as C++) is really just a C `struct` - with some extra features, provided by the language, to make some common tasks easy.

One of the things you'll notice in this project is how common your code is between the two versions. In Java (or Python), we could handle this using abstract data types (that is, two data types that support the same methods). Start thinking about how you might do this in C. (It might be a little awkward in C - but it's certainly **possible!**)

1.2 Separate Compilation

In this project, you will write your first programs that use multiple C files. For each program, your Makefile will need to build two C files - the "main" file, which contains the `main()` function (I'll provide this) and the "funcs" file, which you will write. They will share a header file - which gives the declaration for the struct that they share, and the functions that `main()` can call.

2 Turning In Your Code

Turn in your code using Gradescope. Turn in the following files:

```
intSort_funcs.c
strSort_funcs.c

proj07_strList.h

Makefile
```

Please upload them together as four separate files; do not zip them up into a single zip file or tarball. When Gradescope prompts to upload files, you should select the four files together or drop them together to make sure they all are uploaded.

3 Program 1 - Int Sort

In this program, you will declare various functions which manipulate a linked list that contains `int`s. I have provided a header file which declares the struct, `proj07_intList.h`. I have also provided a C file, which gives the `main()` function for this program: `intSort_main.c` . **Do not change either of these two files. Do not turn them in to Gradescope; if you do, we'll ignore them.**

You will write another C file, `intSort_funcs.c`, and a Makefile to build it all. The Makefile must compile both of the C files (with all of the normal required `gcc` arguments) and then link them together into a single program, called `intSort`.

3.1 Command-Line Arguments

(All of the command-line argument processing is handled by the `main()` function that I've provided.)

This program requires exactly two command-line arguments (not including the program name itself): an input file, and an output file. It will terminate with an error if the input file doesn't exist - but if the output file doesn't exist, it will create it.

The `main()` method will open `FILE*` variables for each file, and pass them to the functions you provide. You will have to read and write these files, as described below.

3.2 Input

The input to this program will be a series of integers, separated by whitespace. `main()` will open this file, and pass it to your function `readInput()` to read.

3.3 Output

This program does not print anything to `stdout`, and will only print to `stderr` in specific circumstances (see `readInput()` below). All of the "regular" output will go to a file, which `main()` will open, and pass to your function `writeOutput()`.

The output from this program is all of the integers read by the program, sorted. Each will be printed along with an index value. **This does not mean that you should create an array!** Instead, your `writeOutput()` will simply count the numbers as it prints them out - and print that count on each line.

3.4 Required Functions

`main()` calls several functions, which you must implement. The prototype for each one is given in the header file. You must implement all of these functions in a file named `intSort_funcs.c`.

You may also define some helper functions if you'd like.

3.4.1 No Globals

Your implementation must not include any global variables. Students often like to save some things - such as the head of a list, or its length - in a global variable. But this is a poor design choice - since your code won't work anymore if we have multiple linked lists in the same program.)

For the same reason, your code **must not** change the fields of the `IntList` struct - or any of the functions called by `main()`. If you do, your code won't compile when we test it, and you will fail all of the testcases.

3.4.2 `readInput()`

This function reads a `FILE*`, and returns an `IntList*`, which is an (already sorted) linked list, containing all of the integers read from the file.

The input in the file will be a set of integers separated by whitespace; you are encouraged to read using `fscanf()` and `%d`. If you encounter any non-numeric input, print an error message to `stderr` and then return a (sorted) list containing the integers that you have read so far. Otherwise, return the list when you hit EOF. (Of course, to "return a list," you will return a pointer to the head of the list.)

The easiest way to sort the list (though very inefficient!) is to simply insert each element as you read it. This takes $O(n^2)$ time overall - since inserting a single element is $O(n)$ - but we'll live with this very

terrible algorithm for this project. If, however, you want to implement a more efficient sorting mechanism, I'll allow it - so long as what you return to `main()` is a **linked list** (no arrays!).

NOTE: If the input file is empty, return `NULL`. This is a **perfectly valid** input - it simply means "empty list."

3.4.3 `getLen()`

This function takes an `IntList*`. It must return the number of elements in the list. **Do this by iterating through the list** - you **must not** save the length in a variable, anywhere in your code.

3.4.4 `writeOutput()`

This function must iterate through the linked list, and write out the integers inside it. See the example executable to figure out the exact format.

3.4.5 `freeList()`

This function must free the entire linked list. **WARNING:** Beginning programmers almost always have a bug in their code, where they free a struct, and then later read the 'next' field. This works just fine, usually - but `valgrind` will complain. What's wrong with this, and how might you fix it?

4 Program 2 - String Sort

This program works more or less the same as the `intSort` program. However, there are several new complexities you must deal with. I have listed those differences below - for everything else, refer to the description of `intSort` above.

4.1 Difference: New `main()`

I have provided a `main()` function for this program. It is almost identical to the one for `intSort`, but has a handful of differences. Use `diff` to see what they are.

4.2 Difference: You Write the Header

This program will use the header `proj07_strList.h`, but I will not provide it! Instead, you must write it (base your code on the `intSort` version).

Since you cannot modify `main()`, you must use names and types which match what `main()` expects. For instance, the struct must be named `StrList`.

4.3 Difference: Storing Strings (a separate `malloc()`)

In `intSort`, each element of the linked list had a `int` field. In `strSort`, these hold `char*` fields. This means that you must `malloc()` buffers to hold the strings! Thus, every word in the file will actually result in **two** `malloc()`d buffers: one for the linked list element, and one for the string itself.

4.4 Difference: Input

The input to this program is a set of strings, separated by whitespace. Since it reads strings, there is no possible way that it will print out an error message (except for `malloc()` failure).

The strings may be of any length. Your program must dynamically allocate the proper amount of memory for each string.

4.5 `readInput()`

This function works mostly like the one from `intSort` - except that each linked list element will hold a string instead of an integer. You must store this string into a second buffer (separate from the linked list element), which you also will `malloc()`.

You must **not** use any C library function or feature which automatically allocates enough space for you. Instead, as with the `manyRecords` program, you must start with a small buffer, and dynamically expand it if you find that it is too small to hold the word that you are reading. (I recommend that you write a helper function to "read one string.")

Your allocation process must obey the following rules:

- You must not allocate more than 8 bytes for the first buffer, including space for the null terminator (so that expansions will be common).
- When expanding, you must never expand the buffer by **more** than doubling the length (for the same reason).
- As with `manyRecords`, you must extend the buffer by calling `malloc()` and `memcpy()`; you **must not** use `realloc()`.

It will be tempting, when you write your code, to use `fscanf()` and `%s`. Will this work? I'd recommend that you use `fgetc()`, and check the characters by hand.

4.6 Error Handling

The `main()` functions that I provide will print to `stderr` (and end with exit code of 1) if the command-line arguments are invalid, or if the input/output files cannot be opened.

Your `readInput()` function should print an error message to `stderr` if the contents of the input file are invalid (this can only happen in `intSort`). However, it should not terminate the program; instead, return the list (as much as you have) to `main()`, and the program will then run normally.

Other than that, the only error conditions are `malloc()` failures.

5 Grading

In this project, we have some new elements which we'll be grading.

- 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 below.

5.1 Testcase Grading

I 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.[1]
- You will lose one-quarter of the credit for the testcase if `valgrind` reports any errors (including failing to `free()` something before your program ended.

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

[1] Remember, we handle `stderr` more loosely than `stdout`. See below.

6 `Makefile` - Requirements

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

Your rule must:

- Automatically rebuild any of the `.o` files (there are four of them!) when the associated C file (or header file) is modified.
- **NOT** rebuild the executable if the C file and header have not been modified.
- Automatically link together the `.o` files into executables when we run

  ```
  make [programName]
  ```

  in the directory. - but only if the `.o` files have been rebuilt!
- 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`).

7 Grading

We have provided compiled versions of these programs for you; you should download them and run them (on Lectura) for comparison. The programs can be copied from `/home/marahman/cs352/projects/proj07/` on Lectura. (The UNIX commands you've learned, such as `ls, cd, cp` will all be useful here.). Note that you can copy the files using the `scp` command. You can find more how to use the `scp` command using the man page. One example to download the files to your current working directory of your non-lectura Linux machine:

```
scp lectura.cs.arizona.edu:/home/marahman/cs352/projects/proj07/* .
```

(We have provided the same on Piazza; however, realize that these will only run on a Linux machine running the x86-64 architecture. If don't have a linux machine, you should work on Lectura and only copy the files to your lectura account and your choice of working directory.)

We have also provided our grading script. (We'll do this for the first couple of assignments, in order to help you understand the requirements - but we will stop doing this as the programs get more complex!) You can copy it from `/home/marahman/cs352/projects/proj06/grade_proj07` on Lectura. A tarball containing all files will be available on Piazza and D2L.

Finally, we have provided a few example inputs to test your program with. (We will use additional ones when we grade your program.) **You should write additional testcases** - try to be creative, and exercise your program in new ways.

NOTE 1: Since this project has multiple programs, it needs testcases for each one. The testcases are named `test_<progName>_*`. In order for the grading script to see your new testcases, please name them according to this standard.

NOTE 2: Some of the programs in this project use `stdin` as their input; name those testcases `test_*.stdin` . Some use command-line arguments; name those testcases `test_*.args` .

7.1 Exact Output

In each Project this semester, most of your grade will come from automatic testing of the code. You must match `stdout` **EXACTLY**, byte for byte. Common mistakes include:

- Extra or missing spaces
- Extra or missing blank lines

- Misspelled words
- Different capitalization

Your code must also give the same return value (0 or 1) as the example executable in every case.

`stderr` will be handled a little more loosely. In each testcase, we will check to see if the standard executable reported **some** error message to `stderr` or not. If it did, then your program must as well; if not, then your program must not print anything, either. However, we will not be comparing the exact error messages.

NOTE: Each testcase either passes, or fails entirely. We do not give partial credit for any testcase - although you may, of course, pass some testcases but not others.

7.2 Running the Grading Script

To run the grading script, arrange your files like this, and then run `./grade_proj07`

```
grade_proj07

example_*

test_*

Makefile

proj07_intList.h   # provided by me
intSort_main.c     # provided by me
intSort_funcs.c

proj07_strList.h
strSort_main.c     # provided by me
strSort_funcs.c
```

The grading script does a number of things:

- Confirms that we have an example executable for each program.
- Confirms that each of the required files has the proper name, in a directory with the proper name. If not, you lose all points for that program.
- Checks that your Makefile works properly.
- Compiles your code. If your code doesn't compile at all, then you lose all points for that program. If it compiles but has warnings, then you will get a heavy deduction.
- Runs your code against all of the testcases. In each case, it runs both your code, and also the example executable. It checks to make sure that both have the same return code - and then also checks to make sure that the outputs match. If not, then it will give you a zero for that testcase. It also checks to make sure that your program runs cleanly under `valgrind`.

  (If you have no testcases for some program, then the script will give you a zero for that program.)

At the end, the grading script reports a score; this score is determined by the number of testcases that you passed - modified for any deductions.

Note: The grading script may show you full points on lectura at it does not have all the testcases. To make sure you pass all the test cases, see gradescope output after submitting there.

7.3 Hand Grading

In addition, each program will be looked at by a human, to check for things which we can't automatically check, like:

- Some of the details of the spec
- Good comments
- Good indentation
- Reasonable variable names

7.4 Point Distribution

In this project, your code will be graded by a script, with your score determined by how many testcases you pass for each program. After that score is calculated, a number of penalties may be applied.

7.4.1 Score Distribution

If any program compiles and runs, but warnings were produced by the compiler, you will lose half your score for that program.

- 5% - `Makefile` works properly
- 30% - `intSort` testcases
- 35% - `strSort` testcases

The last 30% of the score will come from hand grading by the TAs.

8 Standard Requirements

- Your C code should adhere to the coding standards for this class: `https://developer.gnome.org/programming-guidelines/stable/c-coding-style.html.en`
- 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. (Sometimes the error is serious - meaning "the operation cannot be performed." Sometimes, it's more minor, such as "two files are different" or "minor issues happened.")

  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 different 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).

9 `malloc()` Failures

As always, check the return code on `malloc()`, and have some sort of error handling routine for it. However, as we've discussed, it won't be practical to test this code.

10 `free()` all `malloc()`s

Starting with Project 5, you must now `free()` every buffer that you `malloc()`, before your program terminates. If you don't, then `valgrind` will report errors - and you will lose partial credit on your testcase.

This is, of course, a good habit to have! Always keep track of all memory that you have allocated with `malloc()` - and always be responsible to free it.

However, in the Real World, people often ignore the need to free memory, if the program is about to die. As I've mentioned in class, when your process dies, the OS will automatically free **all** of your memory - so it doesn't really matter whether you `free()`d everything or not.

But since we want you to practice `free()`ing your memory, we will require that you `free()` every buffer that you `malloc()`ed - even if you don't free it until right at the **end** of your program.

11 Special Note: `scanf()` and `%s`

Several programs in this class will require you to read strings from `stdin` using `scanf()`. **This can be dangerous, if you read more data than your buffer allows** - since it is possible to read right off the end of the array, and overwrite other memory.

To solve this, `scanf()` allows you to limit the number of characters that you read with the `%s` specifier. **You must always use this feature.** Remember that `scanf()` will also write out a null terminator - so this number **must** be less than the size of your buffer, like this:

```
char buf[128];
int rc = scanf("%127s", buf);
```