CSE 2421 Lab 4

Learning function pointers, timing, and profiling

Due: Monday 3 July 2023 11:59 PM
Early: 2 bonus points if submitted by Saturday 1 July 2023 11:59 PM
Late: Tuesday 4 July 2023 11:59 PM

We'll be using Carmen's timestamps to determine lateness or not. If you are not enrolled, you may send me an email with the zip file. If you are unable to submit by the deadline, you need to send me a screenshot of your finished files with the clock visible and then *still* submit to Carmen when you can.

One of our lovely grad students, Braeden, had a hefty hand in creating this lab. Make sure you read the *excellent* preface he wrote to fully understand Part 1!

This lab contains three parts. Make sure you do all of them!

Something you should do before getting started on the lab is creating your readme file. Simply create a file called "readme" and include your name at the top. As you progress through the lab, you may see different parts (labeled things like **Part 1**). When you see this, write it down in your readme. While in a numbered part of a lab, you'll come across various questions, prepended with a bolded Q and a number (such as **Q5**). When you see this, include the Q and the number, along with you answer to the question(s) in the file. A brief example:

> Name: Braeden J
>
> I certify that I completed all of the work myself with no aid from anyone aside from the instructor or the undergraduate graders.
>
> Part 1:
>
> Q1: My favourite colour is white. Make fun of me.
>
> Q2: While Han did shoot first, prior to the shot being fired Greedo issued a threat in Huttese. This makes Han's actions morally permissible. In this essay I will…

Part 1: Implementing Stack

For this lab, there's a header file provided, as well as some source code. This lab is meant to be a relatively chill (but still interesting) one after the absolute unit that last lab was. Included in the header file is the beginnings of two different implementations of a stack. Your task is to implement both versions of the stack.

The first version of the stack will be using a linked list approach. This is straightforward. Check the values included in the structs for what you need to update. You most likely won't need to modify anything in the header file at all.

The second version will be using a slightly more complicated approach using an array and table doubling. (I expect you to look this up, but TL;DR: when the array is filled up make a new array twice the size and then copy the elements over. When the array reaches a quarter of its current max size create a new array with half the size and copy the elements over.) Upon adding/removing elements, their values are left in the array, but the indices marking the start/end of the structure will change.

Each of these stack implementations are then put inside another structure, named Stack. These contain a pointer to the actual stack as well as some function pointers.

Q1: What very cool word with Latin roots that starts with a P are we implementing here?

Q2: What are at least one advantage and disadvantage of each implementation?

Q3: Prior to doing any timing testing, which approach do you think will be faster?

Q4: When implementing this, why were void pointers used?

Part 2: Timing C programs

Now that the heavy work of implementing those structures is done, let's time them. Construct a test that has plenty of objects being added and/or removed. The number added must be large, otherwise the coarseness of the clock resolution won't let you see meaningful times.

#include <time.h> and make use of the clock() function. You can see information on CPU time here: https://www.gnu.org/software/libc/manual/html_node/CPU-Time.html

Hopefully, you read all of the information on that link, because the very first question relates to something on that page!

Q1: It's possible that the CPU time that we calculate from the above method isn't actually completely correct. Why is that?

In the StackTest() function, calculate the clock time used for each and print it to the console at the end of the execution of the function.

Q2: What is an obvious (but rather silly) way to reduce the CPU time?

Q3: Given your answer to Q2, is CPU time in a vacuum (that is, without any additional context) a good measure of the performance of a program? Why or why not?

Q4: After seeing the results for the test you wrote, which one was faster? Was it the one you predicted? Was there a stark difference in runtime? Why or why not?

Recompile your source files without the -g flag. Take note of the size of the executables, you'll need them for Q7. Consider adding targets to your makefile to allow you to compile with different amounts of optimization!

Q5: Recompile your source files again, but this time without the -g flag and with the -O flag. Rerun them. What happened to the execution times? Why?

Q6: Recompile your source files one more time, but this time without the -g flag and with the -O3 flag. Rerun them. What happened to the execution times? Why?

Q7: Compare the file size of the executables created from Q6 against what you created between Q4 and Q5. Is there a difference? Should we expect there to be?

Q8: You may not have noticed, but the compilation time for your programs increased when you increased the amount that the compiler optimized. Why would this increase in compilation time happen?

Q9: This is a critical thinking question. Look up all the variants of the -O flag. Why do we have so many different optimization flags?

Part 3: Profiling

For the final part of the lab, we'll be profiling with gprof.

All compilation for this part should be done without the -g flag and with the -O and -pg flags.

After compiling and running your program (with only the small change above to the process), you'll notice something interesting in your directory: gmon.out. That's a file created by gprof that holds a bunch of raw data that the gprof program turns into profiling statistics.

Run the command "gprof lab4 > lab4out" replacing lab4 with the name of whatever the executable is that you're profiling and lab4out with whatever you want to name your file that contains the profiling information.

There are two tables inside the generated file, the flat profile and the call graph. The first is not that useful, but the second is quite nice.

Spend some time looking at the call graphs that you've produced. It might take some time to understand exactly what you're looking at. Here's a resource that I think does a wonderful job explaining how to read and understand a call graph:
https://www.math.utah.edu/docs/info/gprof_6.html

Q1: Did you spot anything weird about any of your gprof outputs? If so, what were they and why do you think that weirdness occurred? (The answer to this question could also safely be no.)

Q2: What functions ended up taking up the most time in each of your implementations?

Q3: Do you think there is anything you could do to shrink the time spent in each of those functions? If so, what? (The answer to this question could also safely be no.)

Q4: Think back to your answer to Q2 from Part 2. Does profiling suffer from the same sort of shortcomings that timing suffers from?

Because of your answer to Q4, it's often a good idea to profile more than once. The -s flag with gprof can aid in this.

Q5: Read through the manual page for gprof. Pick out any flag you think sounds interesting/useful and describe why you think it sounds interesting or useful.

When submitting this lab, please include all c files used to produce your executables, any test files and gprof output files you used, your makefile, and your readme.

Part Bonus: Bonus

You may have noticed that your header file contains some things for Queue, in addition to Stack. For a whopping 6 extra credit points (2 per part), you can do the entire lab again but with Queue as your data structure, rather than Stack.