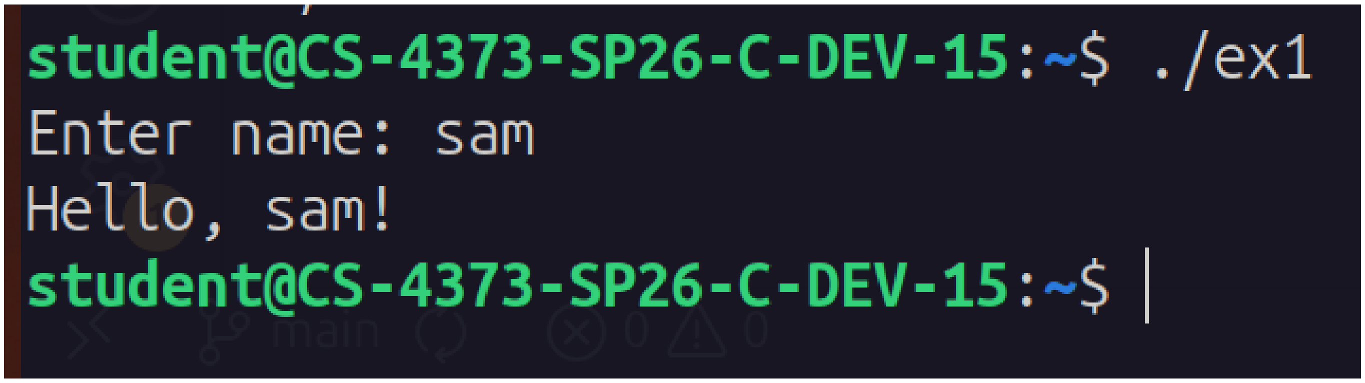
C Exercies

Sam Schulman

Ex 1 - Hello world with stdin

I interpreted the instructions here as requiring taking in the name from stdin, so I tried to use `scanf` since it's quick and dirty and would get the job done.
I encountered a problem, though, when running it on my lab VM in an SSH session.
Something about how SSH handled/buffered newlines made it completely skip waiting for my input and immediately output with an empty name, then exit.
So I switched to using `fgets(stdin)` which worked, although I had to remove an extra newline from the end of the string with some clever array indexing.

Speaking of arrays, I did use static strings for this.

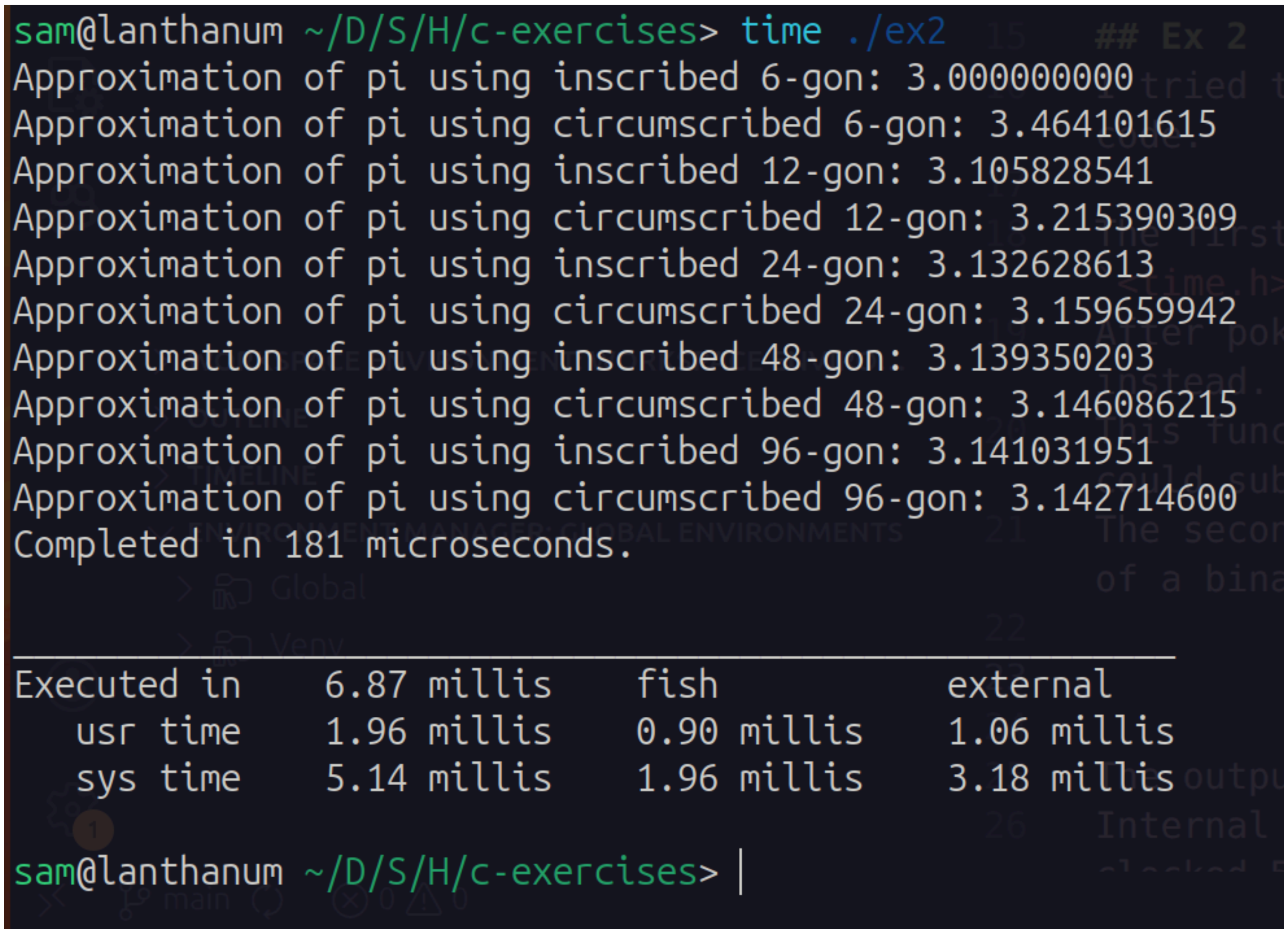


```
student@CS-4373-SP26-C-DEV-15:~$ ./ex1
Enter name: sam
Hello, sam!
student@CS-4373-SP26-C-DEV-15:~$
```

Ex 2 - Archimedes algorithm

I tried two different ways of timing this program: within the code and outside the code.

The first method did give me some trouble, because the ordinary c `time()` from `<time.h>` only gives you time in seconds.
After poking around online I wound up using `<sys/time.h>` and `gettimeofday` instead.
This function returns a struct which has fields for seconds and microseconds, so I could subtract a starting and ending time with adequate resolution.
The second method was with the `time` package on linux, which times the execution of a binary.



```
sam@lanthanum ~/D/S/H/c-exercises> time ./ex2
Approximation of pi using inscribed 6-gon: 3.000000000
Approximation of pi using circumscribed 6-gon: 3.464101615
Approximation of pi using inscribed 12-gon: 3.105828541
Approximation of pi using circumscribed 12-gon: 3.215390309
Approximation of pi using inscribed 24-gon: 3.132628613
Approximation of pi using circumscribed 24-gon: 3.159659942
Approximation of pi using inscribed 48-gon: 3.139350203
Approximation of pi using circumscribed 48-gon: 3.146086215
Approximation of pi using inscribed 96-gon: 3.141031951
Approximation of pi using circumscribed 96-gon: 3.142714600
Completed in 181 microseconds.


________________________________________________________
Executed in    6.87 millis    fish           external
   usr time    1.96 millis    0.90 millis    1.06 millis
   sys time    5.14 millis    1.96 millis    3.18 millis

sam@lanthanum ~/D/S/H/c-exercises> 
```

The output shows the stark contrast in the results.
Internal measurement registered only 181 microseconds and external measurement clocked 38 times that span.
I surmise this is because my internal timing does not count OS-level process initialization or teardown procedures, and that's what makes up the difference.
Further support for this can be found by looking at the CPU time breakdown, since most of the 6.87ms was from sys time, not usr time.

No issues with convergence per se, although in theory I would run up against the limits of the C `double` type, with only so much precision available.
C does have a `long double` type, but the standard library `sqrt` and `pow` functions don't use it, so I stuck with normal `double`.

Ex 3 - Matrix/vector multiplication

I assumed the text file needed to be loaded dynamically and couldn't just be statically programmed into the code, so I wrote a routine using `fgets` and `sscanf` to do that work.

My matrix is a dynamically-allocated buffer, storing the matrix in one-dimensional memory in row-major order.

I didn't want to use a 2-D array because my personal experience with those is that their actual layout in memory can be somewhat counterintuitive and compiler-dependent.

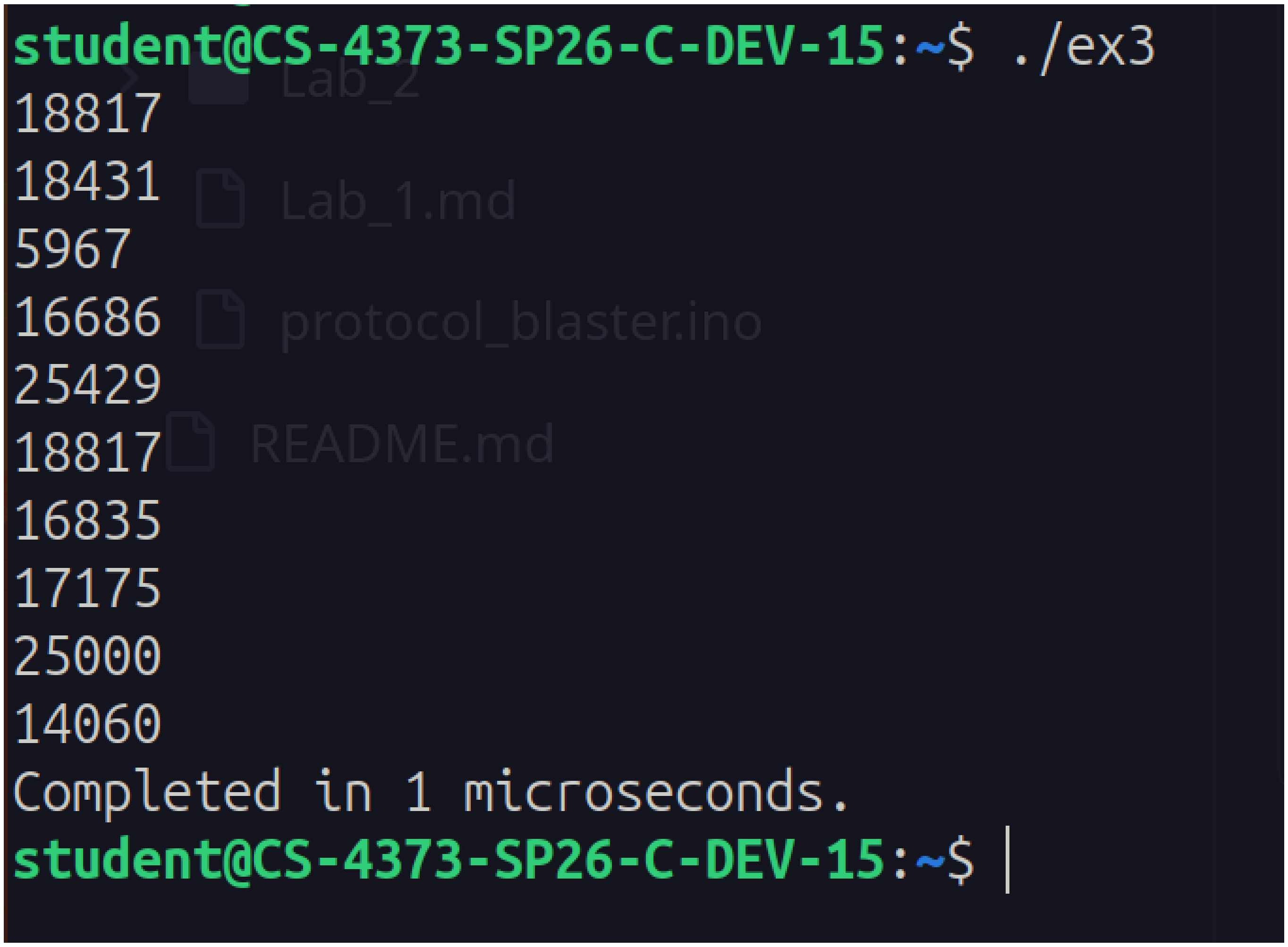
I had to figure out how to read lines from the file with a dynamic number of numbers per line.
I landed on a strategy that involved `sscanf` to load one value at a time, advancing my string pointer to the start of the next value by using `strchr` to find the next space.
I didn't want to spend more time on my I/O than my actual multiplication routine so as a result I'm trusting that the row/column values will not lie to me.
If they did, this would end with garbage data and almost certainly a segfault.

The actual computation was surprisingly simple.
Honestly, it was very satisfying having the real computation be only five lines with the real meat of it being only one.

```
for (int i=0; i < rows; i++){
    for (int j=0; j < cols; j++){
        out[i] += matrix[i*cols+j]*vector[j];
    }
}
```

I wanted to exclude the I/O time from the equation and time only the matrix multiplication part, since that feels like the real core of the task here.
As a result, I decided to stick with my code-internal timing strategy from exercise 2.

Provided below is a screenshot of my output:



```
student@CS-4373-SP26-C-DEV-15:~$ ./ex3
18817
18431
5967
16686
25429
18817
16835
17175
25000
14060
Completed in 1 microseconds.
student@CS-4373-SP26-C-DEV-15:~$ 
```

Remarkably fast computation, though I wonder how it would scale with larger matrices.

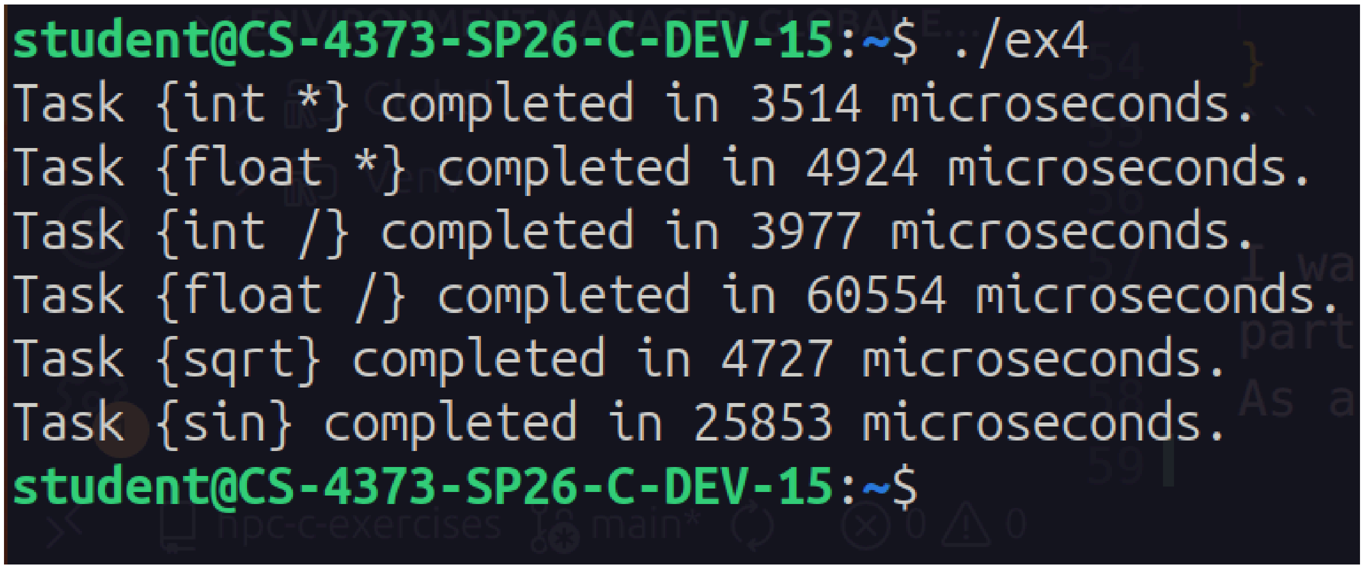
Ex 4 - Timing arithmetic operations

I used a similar timing strategy for this exercise as the previous two, however, since some of these operations are likely on the order of only a few processor cycles each, I figured that even with my timing resolution in the microseconds I would probably be completely unable to time them.
My fix for this is just to run them each one million times (that number is easily tunable by altering my `#define ITER`).

I decided that to really characterize these operations properly, I needed to run them with integer and floating point operands (at least for multiplication and division).
I also needed a strategy for choosing my operands, since doing a ton of meaningless calculations like this is always going to be at the mercy of smart compilers optimizing them into trivial forms or even deleting them altogether.
That introduces considerable potential confusion into this activity, and is why I tried to make my calculations not-obviously-trivial so that the compiler would be more likely to translate them more directly.
This may be overly cautious by me, but oh well.

- **Multiplication**: I used my loop incrementor variable (with either type) and multiplied it with an accumulator variable each time, saving the result to the accumulator. This will certainly result in some overflow, but that doesn't matter because the math isn't the point, the timing is. This is the operation I'm the most worried about being deleted by a clever compiler, and I'm hoping that having the result of one loop's calculation affect the next loop will create a complex enough relationship to confound that.
- **Division**: I selected a pseudorandom divisor between 0 and 999 at program start and divided the loop incrementor variable by that value each time. For the floating point calculation, I used some pointer and type casting tricks to interpret the same *bits* of the integer divisor as floating point, and used that value.
- **Sqrt**: I took the square root of the loop incrementor variable each time.
- **Sin**: I took the sine of the loop incrementor variable each time.

The results:



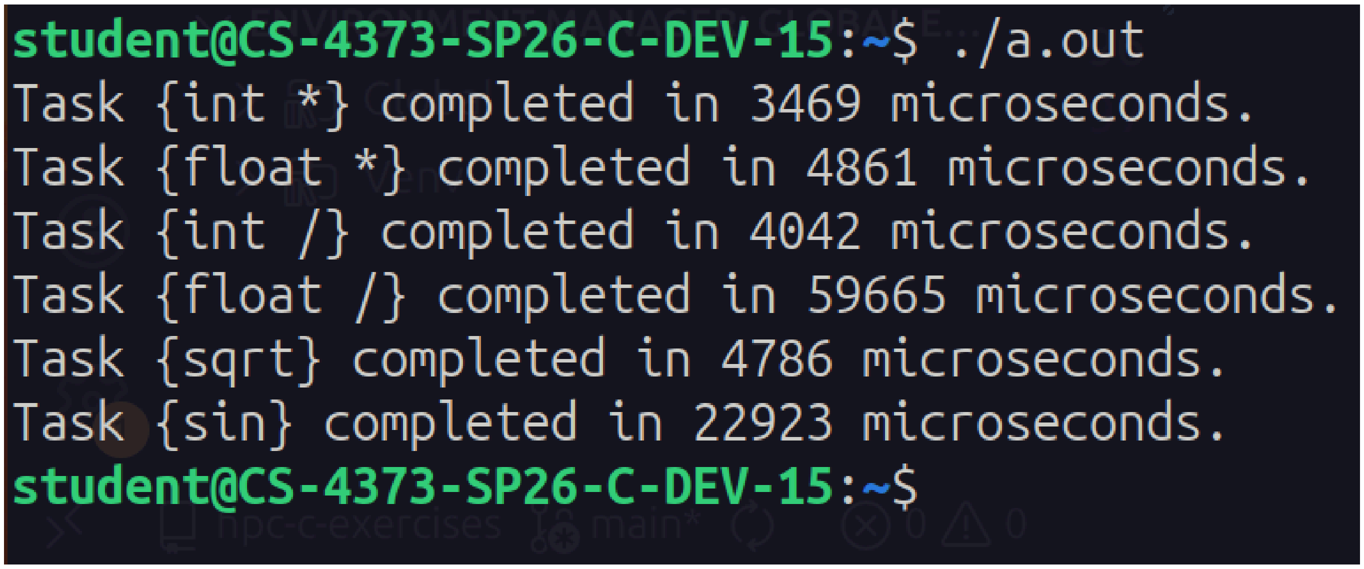
```
student@CS-4373-SP26-C-DEV-15:~$ ./ex4
Task {int *} completed in 3514 microseconds.
Task {float *} completed in 4924 microseconds.
Task {int /} completed in 3977 microseconds.
Task {float /} completed in 60554 microseconds.
Task {sqrt} completed in 4727 microseconds.
Task {sin} completed in 25853 microseconds.
student@CS-4373-SP26-C-DEV-15:~$
```

My iteration seems to have been necessary.
No single one of these operations would have taken more than a microsecond.

Integer multiplication, float multiplication, integer division, and square root are all in the same league.
Square root is a very surprising inclusion in that list, as I would have imagined it to be a more costly iterative process.
I'm very curious as to what's going on there, and if there is any hardware acceleration, or maybe precomputed values for low numbers which are more likely to be queried.

Next on the list is sine, which is roughly 5-6 times slower than these, and then floating point division, which is more than twice as slow as sine.
I knew floating point division was costly, but I would have expected the math library functions to beat it handily on speed.

Out of sheer curiosity, I re-compiled this code with a different compiler (clang) to see if that made a difference.
It didn't.
The timing values are barely different at all:

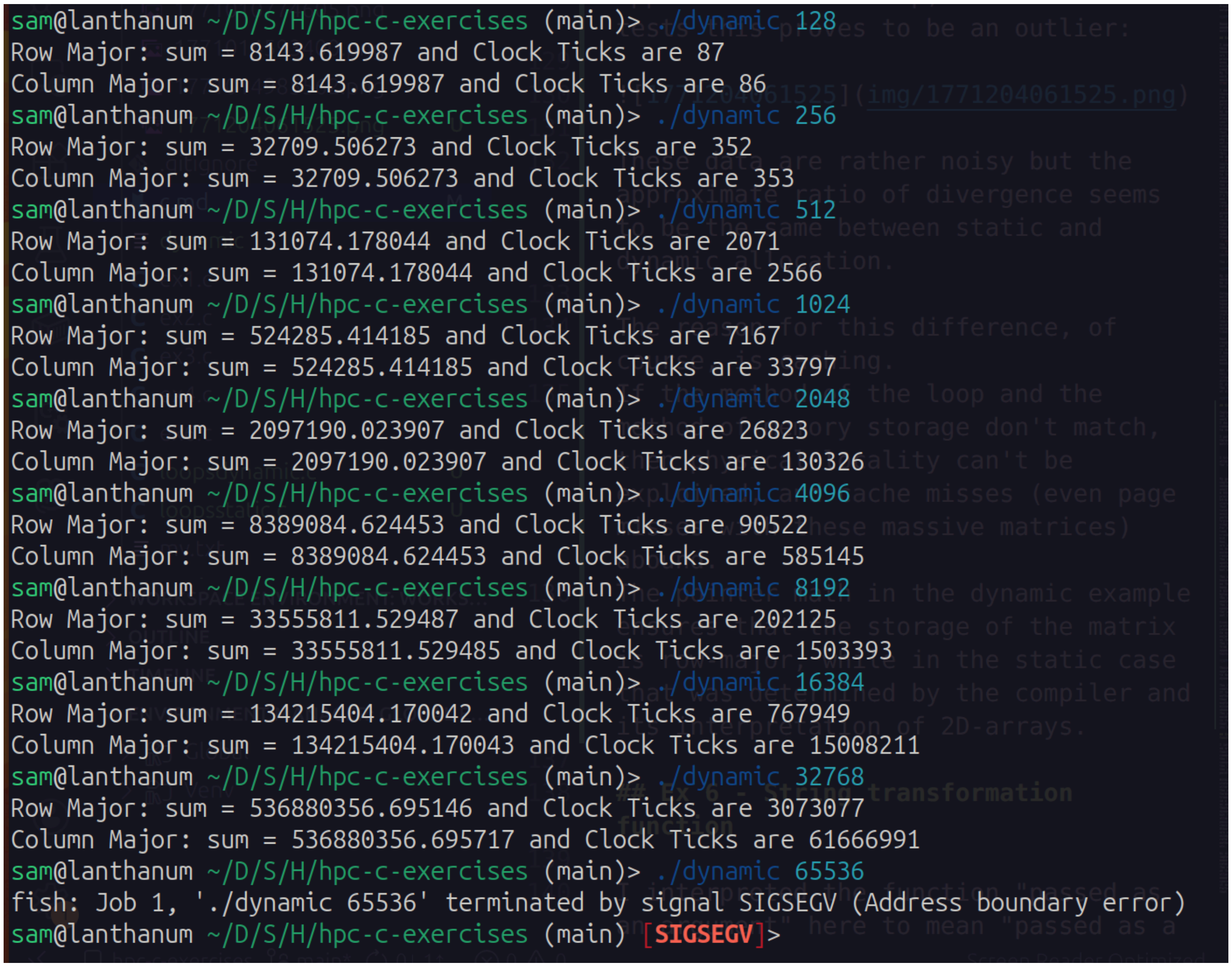


```
student@CS-4373-SP26-C-DEV-15:~$ ./a.out
Task {int *} completed in 3469 microseconds.
Task {float *} completed in 4861 microseconds.
Task {int /} completed in 4042 microseconds.
Task {float /} completed in 59665 microseconds.
Task {sqrt} completed in 4786 microseconds.
Task {sin} completed in 22923 microseconds.
student@CS-4373-SP26-C-DEV-15:~$
```

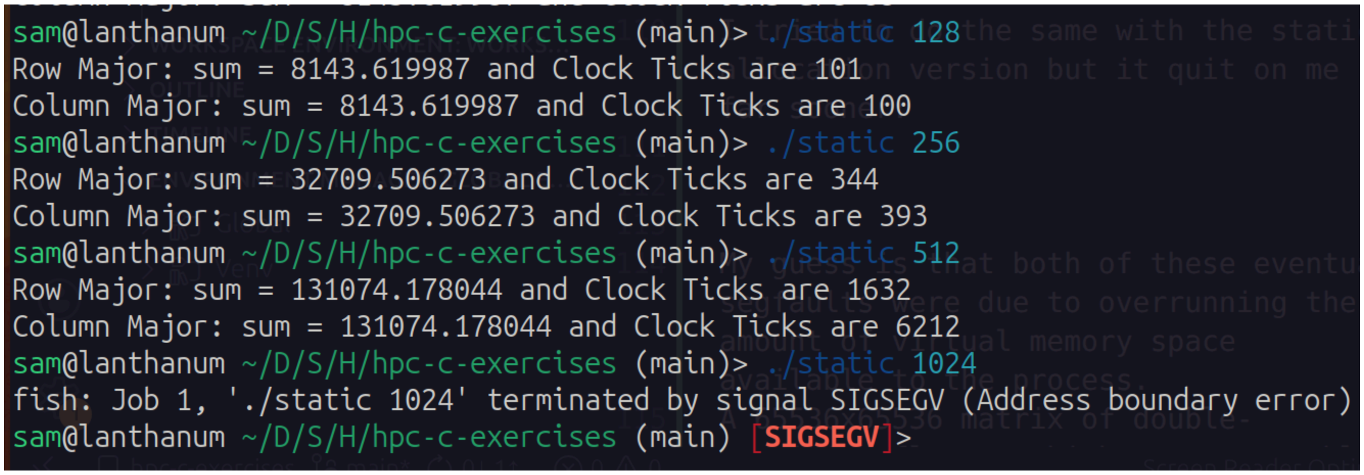
Ex 5 - Row-major vs Column-major

Implementing the column-major work here was simple, just copying the row-major work and swapping the loop headers.
I decided to make `n` determined by a command line argument to make it easier to vary without recompiling.

I put the dynamic-allocation version through its paces with escalating matrix sizes, until my system rebelled:


```
sam@lanthanum ~/D/S/H/hpc-c-exercises (main)> ./dynamic 128
Row Major: sum = 8143.619987 and Clock Ticks are 87
Column Major: sum = 8143.619987 and Clock Ticks are 86
sam@lanthanum ~/D/S/H/hpc-c-exercises (main)> ./dynamic 256
Row Major: sum = 32709.506273 and Clock Ticks are 352
Column Major: sum = 32709.506273 and Clock Ticks are 353
sam@lanthanum ~/D/S/H/hpc-c-exercises (main)> ./dynamic 512
Row Major: sum = 131074.178044 and Clock Ticks are 2071
Column Major: sum = 131074.178044 and Clock Ticks are 2566
sam@lanthanum ~/D/S/H/hpc-c-exercises (main)> ./dynamic 1024
Row Major: sum = 524285.414185 and Clock Ticks are 7167
Column Major: sum = 524285.414185 and Clock Ticks are 33797
sam@lanthanum ~/D/S/H/hpc-c-exercises (main)> ./dynamic 2048
Row Major: sum = 2097190.023907 and Clock Ticks are 26823
Column Major: sum = 2097190.023907 and Clock Ticks are 130326
sam@lanthanum ~/D/S/H/hpc-c-exercises (main)> ./dynamic 4096
Row Major: sum = 8389084.624453 and Clock Ticks are 90522
Column Major: sum = 8389084.624453 and Clock Ticks are 585145
sam@lanthanum ~/D/S/H/hpc-c-exercises (main)> ./dynamic 8192
Row Major: sum = 33555811.529487 and Clock Ticks are 202125
Column Major: sum = 33555811.529485 and Clock Ticks are 1503393
sam@lanthanum ~/D/S/H/hpc-c-exercises (main)> ./dynamic 16384
Row Major: sum = 134215404.170042 and Clock Ticks are 767949
Column Major: sum = 134215404.170043 and Clock Ticks are 15008211
sam@lanthanum ~/D/S/H/hpc-c-exercises (main)> ./dynamic 32768
Row Major: sum = 536880356.695146 and Clock Ticks are 3073077
Column Major: sum = 536880356.695717 and Clock Ticks are 61666991
sam@lanthanum ~/D/S/H/hpc-c-exercises (main)> ./dynamic 65536
fish: Job 1, './dynamic 65536' terminated by signal SIGSEGV (Address boundary error)
sam@lanthanum ~/D/S/H/hpc-c-exercises (main) [SIGSEGV]>
```

I tried to do the same with the static-allocation version but it quit on me far sooner:


```
sam@lanthanum ~/D/S/H/hpc-c-exercises (main)> ./static 128
Row Major: sum = 8143.619987 and Clock Ticks are 101
Column Major: sum = 8143.619987 and Clock Ticks are 100
sam@lanthanum ~/D/S/H/hpc-c-exercises (main)> ./static 256
Row Major: sum = 32709.506273 and Clock Ticks are 344
Column Major: sum = 32709.506273 and Clock Ticks are 393
sam@lanthanum ~/D/S/H/hpc-c-exercises (main)> ./static 512
Row Major: sum = 131074.178044 and Clock Ticks are 1632
Column Major: sum = 131074.178044 and Clock Ticks are 6212
sam@lanthanum ~/D/S/H/hpc-c-exercises (main)> ./static 1024
fish: Job 1, './static 1024' terminated by signal SIGSEGV (Address boundary error)
sam@lanthanum ~/D/S/H/hpc-c-exercises (main) [SIGSEGV]>
```

My guess is that both of these eventual segfaults were due to overrunning the amount of virtual memory space available to the process.
A 65536x65536 matrix of double-precision floats, which are presumably 64-bit (8-byte), requires 2^35 bytes of storage.
That is 32 GiB, which is roughly the total amount of physical memory available on the machine I ran this particular exercise on, and we were trying to use that much in only a single segment of

memory (the heap).
For contrast, the previous step, which took over a minute to run but didn't crash, would only be 2^33 bytes, or 8 GiB.
Now, virtual memory and physical memory are not the same thing and the former is very much intended to be able to exceed the latter, but only within reason.
The arbiter of reason here is the operating system, and I don't know where the linux kernel draws the line, but apparently this was over it.

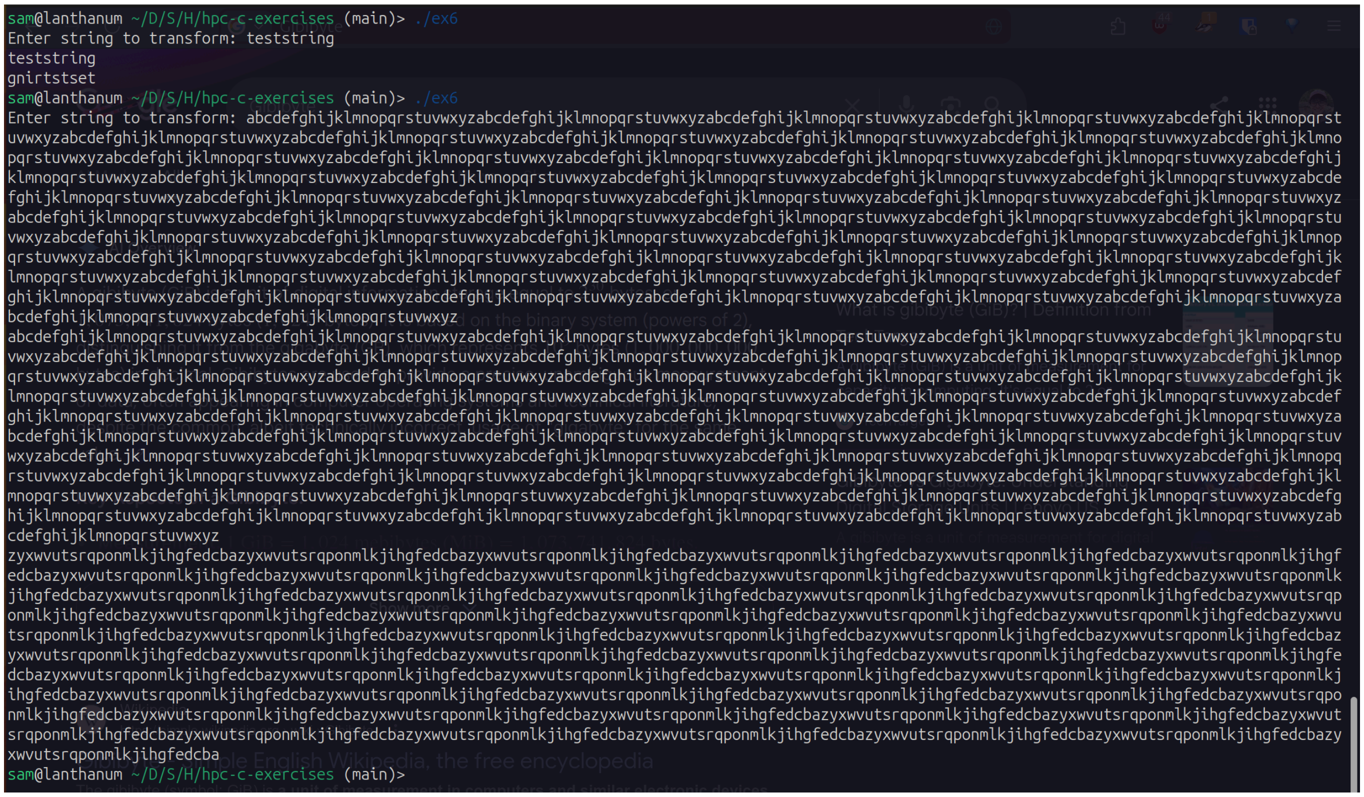
The static version failed sooner because there was less available space on the stack than there was on the heap.
A 512x512 matrix consumes only 2^21 bytes or 2 MiB.
The step which crashed the program would have been 8 MiB.
The stack just doesn't get the kind of space the heap does, because it's meant for normal function activation records, not colossal data structures.

Examining the timing data, while it's pretty close at first between row- and column-major, they begin to diverge.
In my final dynamic run, the column-major loop took around 20 times as long as the row-major loop.
These data are rather noisy but the approximate ratio of divergence seems to be the same between static and dynamic allocation.

The reason for this difference, of course, is caching.
If the method of the loop and the method of memory storage don't match, then physical locality can't be exploited, and cache misses (even page misses with these massive matrices) abound.
The pointer math in the dynamic example ensures that the storage of the matrix is row-major, while in the static case that was determined by the compiler and its interpretation of 2D-arrays.

Ex 6 - String transformation function

I interpreted the function "passed as an argument" here to mean "passed as a function argument" rather than a command-line argument, mostly because the latter doesn't really make any sense, and also because reading between the lines on this assignment I think the purpose is to make us use function pointers.



```
sam@lanthanum ~/D/S/H/hpc-c-exercises (main)> ./ex6
Enter string to transform: teststring
teststring
gnirtstset
sam@lanthanum ~/D/S/H/hpc-c-exercises (main)> ./ex6
Enter string to transform: abcdefghijklmnopqrstuvwxyzabcdefghijklmnopqrstuvwxyzabcdefghijklmnopqrstuvwxyzabcdefghijklmnopqrstuvwxyzabcdefghijklmnopqrst
uvwxyzabcdefghijklmnopqrstuvwxyzabcdefghijklmnopqrstuvwxyzabcdefghijklmnopqrstuvwxyzabcdefghijklmnopqrstuvwxyzabcdefghijklmnopqrstuvwxyzabcdefghijklmno
pqrstuvwxyzabcdefghijklmnopqrstuvwxyzabcdefghijklmnopqrstuvwxyzabcdefghijklmnopqrstuvwxyzabcdefghijklmnopqrstuvwxyzabcdefghijklmnopqrstuvwxyzabcdefghij
klmnopqrstuvwxyzabcdefghijklmnopqrstuvwxyzabcdefghijklmnopqrstuvwxyzabcdefghijklmnopqrstuvwxyzabcdefghijklmnopqrstuvwxyzabcdefghijklmnopqrstuvwxyzabcde
fghijklmnopqrstuvwxyzabcdefghijklmnopqrstuvwxyzabcdefghijklmnopqrstuvwxyzabcdefghijklmnopqrstuvwxyzabcdefghijklmnopqrstuvwxyzabcdefghijklmnopqrstuvwxyz
abcdefghijklmnopqrstuvwxyzabcdefghijklmnopqrstuvwxyzabcdefghijklmnopqrstuvwxyzabcdefghijklmnopqrstuvwxyzabcdefghijklmnopqrstuvwxyzabcdefghijklmnopqrstu
vwxyzabcdefghijklmnopqrstuvwxyzabcdefghijklmnopqrstuvwxyzabcdefghijklmnopqrstuvwxyzabcdefghijklmnopqrstuvwxyzabcdefghijklmnopqrstuvwxyzabcdefghijklmnop
qrstuvwxyzabcdefghijklmnopqrstuvwxyzabcdefghijklmnopqrstuvwxyzabcdefghijklmnopqrstuvwxyzabcdefghijklmnopqrstuvwxyzabcdefghijklmnopqrstuvwxyzabcdefghijk
lmnopqrstuvwxyzabcdefghijklmnopqrstuvwxyzabcdefghijklmnopqrstuvwxyzabcdefghijklmnopqrstuvwxyzabcdefghijklmnopqrstuvwxyzabcdefghijklmnopqrstuvwxyzabcdef
ghijklmnopqrstuvwxyzabcdefghijklmnopqrstuvwxyzabcdefghijklmnopqrstuvwxyzabcdefghijklmnopqrstuvwxyzabcdefghijklmnopqrstuvwxyzabcdefghijklmnopqrstuvwxyza
bcdefghijklmnopqrstuvwxyzabcdefghijklmnopqrstuvwxyz
abcdefghijklmnopqrstuvwxyzabcdefghijklmnopqrstuvwxyzabcdefghijklmnopqrstuvwxyzabcdefghijklmnopqrstuvwxyzabcdefghijklmnopqrstuvwxyzabcdefghijklmnopqrstu
vwxyzabcdefghijklmnopqrstuvwxyzabcdefghijklmnopqrstuvwxyzabcdefghijklmnopqrstuvwxyzabcdefghijklmnopqrstuvwxyzabcdefghijklmnopqrstuvwxyzabcdefghijklmnop
qrstuvwxyzabcdefghijklmnopqrstuvwxyzabcdefghijklmnopqrstuvwxyzabcdefghijklmnopqrstuvwxyzabcdefghijklmnopqrstuvwxyzabcdefghijklmnopqrstuvwxyzabcdefghijk
lmnopqrstuvwxyzabcdefghijklmnopqrstuvwxyzabcdefghijklmnopqrstuvwxyzabcdefghijklmnopqrstuvwxyzabcdefghijklmnopqrstuvwxyzabcdefghijklmnopqrstuvwxyzabcdef
ghijklmnopqrstuvwxyzabcdefghijklmnopqrstuvwxyzabcdefghijklmnopqrstuvwxyzabcdefghijklmnopqrstuvwxyzabcdefghijklmnopqrstuvwxyzabcdefghijklmnopqrstuvwxyza
bcdefghijklmnopqrstuvwxyzabcdefghijklmnopqrstuvwxyzabcdefghijklmnopqrstuvwxyzabcdefghijklmnopqrstuvwxyzabcdefghijklmnopqrstuvwxyzabcdefghijklmnopqrstuv
wxyzabcdefghijklmnopqrstuvwxyzabcdefghijklmnopqrstuvwxyzabcdefghijklmnopqrstuvwxyzabcdefghijklmnopqrstuvwxyzabcdefghijklmnopqrstuvwxyzabcdefghijklmnopq
rstuvwxyzabcdefghijklmnopqrstuvwxyzabcdefghijklmnopqrstuvwxyzabcdefghijklmnopqrstuvwxyzabcdefghijklmnopqrstuvwxyzabcdefghijklmnopqrstuvwxyzabcdefghijkl
mnopqrstuvwxyzabcdefghijklmnopqrstuvwxyzabcdefghijklmnopqrstuvwxyzabcdefghijklmnopqrstuvwxyzabcdefghijklmnopqrstuvwxyzabcdefghijklmnopqrstuvwxyzabcdefg
hijklmnopqrstuvwxyzabcdefghijklmnopqrstuvwxyzabcdefghijklmnopqrstuvwxyzabcdefghijklmnopqrstuvwxyzabcdefghijklmnopqrstuvwxyzabcdefghijklmnopqrstuvwxyzab
cdefghijklmnopqrstuvwxyz
zyxwvutsrqponmlkjihgfedcbazyxwvutsrqponmlkjihgfedcbazyxwvutsrqponmlkjihgfedcbazyxwvutsrqponmlkjihgfedcbazyxwvutsrqponmlkjihgfedcbazyxwvutsrqponmlkjihgf
edcbazyxwvutsrqponmlkjihgfedcbazyxwvutsrqponmlkjihgfedcbazyxwvutsrqponmlkjihgfedcbazyxwvutsrqponmlkjihgfedcbazyxwvutsrqponmlkjihgfedcbazyxwvutsrqponmlk
jihgfedcbazyxwvutsrqponmlkjihgfedcbazyxwvutsrqponmlkjihgfedcbazyxwvutsrqponmlkjihgfedcbazyxwvutsrqponmlkjihgfedcbazyxwvutsrqponmlkjihgfedcbazyxwvutsrqp
onmlkjihgfedcbazyxwvutsrqponmlkjihgfedcbazyxwvutsrqponmlkjihgfedcbazyxwvutsrqponmlkjihgfedcbazyxwvutsrqponmlkjihgfedcbazyxwvutsrqponmlkjihgfedcbazyxwvu
tsrqponmlkjihgfedcbazyxwvutsrqponmlkjihgfedcbazyxwvutsrqponmlkjihgfedcbazyxwvutsrqponmlkjihgfedcbazyxwvutsrqponmlkjihgfedcbazyxwvutsrqponmlkjihgfedcbaz
yxwvutsrqponmlkjihgfedcbazyxwvutsrqponmlkjihgfedcbazyxwvutsrqponmlkjihgfedcbazyxwvutsrqponmlkjihgfedcbazyxwvutsrqponmlkjihgfedcbazyxwvutsrqponmlkjihgfe
dcbazyxwvutsrqponmlkjihgfedcbazyxwvutsrqponmlkjihgfedcbazyxwvutsrqponmlkjihgfedcbazyxwvutsrqponmlkjihgfedcbazyxwvutsrqponmlkjihgfedcbazyxwvutsrqponmlkj
ihgfedcbazyxwvutsrqponmlkjihgfedcbazyxwvutsrqponmlkjihgfedcbazyxwvutsrqponmlkjihgfedcbazyxwvutsrqponmlkjihgfedcbazyxwvutsrqponmlkjihgfedcbazyxwvutsrqpo
nmlkjihgfedcbazyxwvutsrqponmlkjihgfedcbazyxwvutsrqponmlkjihgfedcbazyxwvutsrqponmlkjihgfedcbazyxwvutsrqponmlkjihgfedcbazyxwvutsrqponmlkjihgfedcbazyxwvut
srqponmlkjihgfedcbazyxwvutsrqponmlkjihgfedcbazyxwvutsrqponmlkjihgfedcbazyxwvutsrqponmlkjihgfedcbazyxwvutsrqponmlkjihgfedcbazyxwvutsrqponmlkjihgfedcbazy
xwvutsrqponmlkjihgfedcba
sam@lanthanum ~/D/S/H/hpc-c-exercises (main)>
```

Within the scope of the problem as stated, there aren't a lot of alternative strategies to consider.
The problem essentially specifies the format of main function calling transform-applying function which takes in a specific transformation function as an argument.
There's an argument to be made about dynamic v.s. static strings, perhaps, but following the dictum to use dynamic allocation and pointer math where possible/appropriate, there's not much choice to be made there.
Other possible differences of approach are eliminated on a similar basis: using array syntax or pointer math to reference individual string characters, method of getting string from stdin, etc.

As with any pointer-heavy implementation, proper tracking of memory is essential here to avoid a catastrophe.
The dynamic string must be lengthened at the appropriate time, which entails allocating new memory, ensuring the old memory has been deallocated if the new space has moved, tracking the new size, it must be freed upon exit, etc.
Fortunately, `realloc` handles some of that for us.