C-mini-challenges

Open in Visual Studio Code

For the following Mini-Challenges, you may use the Ubuntu VM set up for you. You will need to install gcc (and maybe valgrind), e.g., "sudo apt install gcc." The Cyber VM Infrastructure Notes should guide you on how to get started with the VMs.

I recommend creating a SSH key for the VM and associating it with your GitHub account. It takes a little bit of work but should be worth it in the long run. Guidance on how SSH key generation and GitHub association - generated by Claude and vetted by me - is available in the course contents section for this class.

And now, on to the Mini-Challenges:

1. Print “Hello, NAME” where NAME is input from the keyboard.
2. Implement Archimedes algorithm to estimate pi for inscribed/circumscribed polygons with n sides, up to 100, doubling n at each step, and time it.
3. Implement matrix – vector multiplication. Read in the following text file (mv.txt) which contains the matrix and vector to be multiplied. Print your answer to the screen and time the computation. The format of mv.txt is: line 1 contains numrows, numcols. The next numrows contains the rows of the matrix, each with numcols integer values. The next line contains the length of the vector. The next line contains the vector of that length.
4. Compare the speed of *,/,sqrt, sin operations/functions.
5. Use the attached code snippets as a basis for comparing the performance of row-major vs. column major computations. One snippet uses a static allocation for the array, the other allocates the array dynamically. Do a little experimentation with each approach. Vary the size of the square array from 128 X 128 on up, doubling it in size each time. Chart your results. Is there a difference in performance or behavior between static and dynamic? Between row-major and column-major? In terms of the latter, valgrind has been installed on your Linux VM, and its cachegrind tool facility may help provide some insights. Do some OSINT research to learn about valgrind....
6. Write a program that accepts a string input from stdio and sends it to a function that transforms it according a transposition function passed in to it as an argument. The function will print out the string, transform it, and then print out the result. The transposition function, you can assume, simply shuffles the existing characters in the string. Build a transposition function that reverses the string and apply it. Where appropriate and possible, use dynamic allocation and pointer arithmetic to get the job done.

C-Mini-Challenges Ouptut

```
(base) mikeyferguson@MacBook-Air-91 c-mini-challenges-Pirate-Hunter-Zoro %
gcc exercises.c -lm -o bin/main
(base) mikeyferguson@MacBook-Air-91 c-mini-challenges-Pirate-Hunter-Zoro %
```

```
./bin/main
Enter a name:
Mikey
Hello, Mikey!


============================================================
n: 128
C: 3.142224
I: 3.141277

Elapsed 0.000005 seconds...


============================================================
Resulting Product:
18817.000000 18431.000000 5967.000000 16686.000000 25429.000000
18817.000000 16835.000000 17175.000000 25000.000000 14060.000000
Time Elapsed: 0.000004 seconds


============================================================
Division Time for 10000000 Operations: 0.019092 seconds
Multiplication Time for 10000000 Operations: 0.011947 seconds
Square Root Time for 10000000 Operations: 0.007973 seconds
Sine Time for 10000000 Operations: 0.047004 seconds


============================================================
Static Array Size 128 by 128 Row Major Time: 0.000030 Seconds
Static Array Size 128 by 128 Column Major Time: 0.000012 Seconds
Dynamic Array Size 128 by 128 Row Major Time: 0.000030 Seconds
Dynamic Array Size 128 by 128 Column Major Time: 0.000014 Seconds

Static Array Size 256 by 256 Row Major Time: 0.000105 Seconds
Static Array Size 256 by 256 Column Major Time: 0.000067 Seconds
Dynamic Array Size 256 by 256 Row Major Time: 0.000092 Seconds
Dynamic Array Size 256 by 256 Column Major Time: 0.000062 Seconds

Static Array Size 512 by 512 Row Major Time: 0.000354 Seconds
Static Array Size 512 by 512 Column Major Time: 0.000242 Seconds
Dynamic Array Size 512 by 512 Row Major Time: 0.000360 Seconds
Dynamic Array Size 512 by 512 Column Major Time: 0.000247 Seconds


============================================================
Enter a string to be reversed:
terrestrial
Original String: terrestrial

Reverse String: lairtserret
```

C-Mini-Challenges Responses

1. For the Hello Name challenge

a. Did you try passing your name as an argument from the command line or did you use scanf? Why? I used `scanf` because it allows for an interactive prompt, which makes the program slightly easier to use for a casual user compared to passing command-line arguments via `argv`. However, in a strict High-Performance Computing environment, command-line arguments are generally preferred for automation.

b. How did you manage or allocate the strings? (Static or dynamic) I allocated the string statically by declaring a character array of a fixed size (`char name[100];`) on the stack.

2. For Archimedes algorithm

a. How did you time your program? I timed the program using the `clock_t` data type and the `clock()` function from the `<time.h>` library to record the start and end cycles. I then divided the difference by the `CLOCKS_PER_SEC` constant to calculate the elapsed time in seconds.

b. Were there any issues with precision and/or convergence that you noticed? Within the bounds of the assignment (stopping at n=128), there were no noticeable convergence issues; the approximation of pi became steadily more accurate. However, if the iterations were pushed significantly higher, floating-point precision limits and catastrophic cancellation would eventually cause the approximation to degrade.

3. For Matrix-vector multiplication

a. How did you allocate and access your matrix? I dynamically allocated a flattened 1D array on the heap using `malloc`. To access the equivalent of a 2D `matrix[i][j]` position, I calculated the offset using `i * cols + j`.

b. Were there any challenges in reading in the file? Once the correct standard library calls (`fopen`, `fscanf`) were implemented, reading the file was straightforward. The main challenge was ensuring proper validation, such as checking that the read vector length strictly matched the matrix column count before proceeding.

c. Was there anything special about the actual computation? To optimize the computation, I processed the dot products sequentially but used a local accumulator variable (`sum`) for the inner loop.

d. What was your strategy for timing? I used `clock_t` and `clock()` again, ensuring that the timer only wrapped the mathematical computation loops, entirely excluding the slow file I/O operations from the measurement.

4. For measuring the speed of arithmetic computations

a. What was your timing strategy? Because modern CPUs execute single arithmetic instructions too quickly to measure accurately, I placed the operation inside a loop running 10,000,000 times and timed the entire loop. I assigned the result to a `volatile double` variable which was necessary to prevent the compiler from optimizing and skipping the loop entirely, since the result is never printed or used.

b. Are all arithmetic operations created equal? No. Based on the timing results, the operations from fastest to slowest were: square root, multiplication, division, and sine. Square root operations are heavily optimized at the hardware instruction level, making them unexpectedly fast compared to standard software assumptions.

5. For the row-major/column-major exercise

a. What did you observe about differences in program behavior in static vs dynamic allocation of arrays, and how do you explain it? For small arrays, static allocation was slightly faster due to the speed of stack memory access. However, the static Variable-Length Array (VLA) quickly hit the memory limits of the stack. For larger sizes (e.g., 512x512 and above), the dynamically allocated 1D flat array on the heap proved more stable and faster, as it avoided the overhead of complex stack boundary calculations.

b. What did you observe about differences in program behavior in row-major vs. column major computations and how do you explain it? C stores 2D arrays in row-major order (contiguous memory). Row-major traversal is theoretically much faster because it utilizes spatial locality; pulling one value loads the adjacent values into the CPU's L1 cache, resulting in fewer cache misses. In my case, column-major appeared faster or comparable at very small array sizes, which was likely an anomaly caused by aggressive hardware prefetching or microsecond clock noise. At larger scales, row-major access is fundamentally more cache-efficient.

6. For the string transform problem

a. What were some alternative implementation strategies could you entertain here? An alternative implementation could involve utilizing custom data structures, such as implementing a stack and queue from scratch. The characters could be pushed onto the stack sequentially and then popped off to reverse the order, whereas the character pointers could be pushed onto the queue and dequeued to preserve their order.

b. What programming hazards/pitfalls should be considered in your general approach? When manipulating strings via pointer arithmetic, hazards include failing to account for the null terminator (`\0`), accidentally swapping the newline character (`\n`) to the front of the string, or decrementing a pointer past the starting address of the array if the user inputs an empty string.

Note to self on how I should compile

```
gcc exercises.c -o bin/program_name
```