Mini-C Challenges Submission

Hello, here is where you will see my C submissions!

Part 1

Print “Hello, NAME” where NAME is input from the keyboard.

In [ ]:
```
#include <stdio.h>

int main() {
    char fullName[30];

    printf("Type your full name: \n");

    fgets(fullName, sizeof(fullName), stdin);

    printf("Hello %s", fullName);

    return 0;
}
// I used fgets because it says to use keyboard input in the instructions an
```

$ gcc part1.c -o part1 && ./part1 Type your full name: Trevor Hurst Hello Trevor Hurst

Part 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.

> I used `$ sudo perf stat ./part2` for timing in this case. Yes, you can use a clock_t variable and time.h but, I didn't want the timing to mess with the code execution, perf stat also gives us more performance statistics

In [ ]:
```
#include <stdio.h>
#include <math.h>

int sides = 3;
float len_inner = 0.8660254; // 0.5*sqrt(3);
float len_outer = 1.7320508; // sqrt(3);
float perimeter_inner;
float perimeter_outer;

int main(){
        while (sides < 100){
```

```
                perimeter_outer = len_outer*sides;
                perimeter_inner = len_inner*sides;

                len_outer = len_outer / (1 + sqrt(1 + len_outer * len_outer)
                len_inner = sqrt(0.5-0.5*sqrt(1 - len_inner * len_inner));

                sides*= 2;
        }
        printf("pi bounds: %f <= pi <= %f\n", perimeter_inner, perimeter_out
}
```

$ sudo perf stat ./part2 pi bounds: 3.141038 <= pi <= 3.142715 Performance counter stats for './part2': 0.33 msec task-clock # 0.334 CPUs utilized 0 context-switches # 0.000 /sec 0 cpu-migrations # 0.000 /sec 70 page-faults # 213.389 K/sec 1,212,428 cycles # 3.696 GHz 26,629 stalled-cycles-frontend # 2.20% frontend cycles idle 232,277 stalled-cycles-backend # 19.16% backend cycles idle 775,008 instructions # 0.64 insn per cycle # 0.30 stalled cycles per insn 176,379 branches # 537.675 M/sec <not counted> branch-misses (0.00%) 0.000983511 seconds time elapsed 0.001228000 seconds user 0.000000000 seconds sys

Part 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.

In [3]:

```
#include <stdio.h>
#include <string.h>
#include <math.h>


int main(){

        FILE *fptr;
        int numrows, numcols;
        fptr = fopen("mv.txt", "r");

        fscanf(fptr, "%d %d", &numrows, &numcols);

        int matrix[numrows][numcols];
        for (int i = 0; i < numrows; i++){
                for(int j = 0; j < numcols; j++){
                        fscanf(fptr, "%d", &matrix[i][j]);
                }
        }

        int vectorlen;
        fscanf(fptr, "%d", &vectorlen);
        int vector[vectorlen];

        for (int i = 0; i < vectorlen; i++){
                fscanf(fptr, "%d", &vector[i]);
```

```
        }

        int result[numrows];
        memset(result, 0, sizeof(result));

        for (int i = 0; i < numrows; i++){
                for (int j = 0; j < numcols; j++){
                        result[i]+=vector[j]*matrix[i][j];
                }
        }

        for (int i = 0; i < numrows; i++){
                printf("%d ", result[i]);
        }
        fclose(fptr);

        return 0;

}
```

```
18817 18431 5967 16686 25429 18817 16835 17175 25000 14060
```

$ sudo perf stat ./part3 18817 18431 5967 16686 25429 18817 16835 17175 25000 14060 Performance counter stats for './part3': 0.30 msec task-clock # 0.297 CPUs utilized 0 context-switches # 0.000 /sec 0 cpu-migrations # 0.000 /sec 60 page-faults # 200.294 K/sec 1,106,680 cycles # 3.694 GHz 23,867 stalled-cycles-frontend # 2.16% frontend cycles idle 209,091 stalled-cycles-backend # 18.89% backend cycles idle 740,057 instructions # 0.67 insn per cycle # 0.28 stalled cycles per insn 165,650 branches # 552.978 M/sec <not counted> branch-misses (0.00%) 0.001007680 seconds time elapsed 0.001234000 seconds user 0.000000000 seconds sys

Part 4

Compare the speed of *,/,sqrt, sin operations/functions.

> I am going to make multiple programs for this part, that way I can get the nice cool performance statistics for my code!

I decided to go with multiplying and dividing by 3 and making sure to turn off compiler optimizations so we're comparing apples-to-apples.

I had been doing i*i and i/i but, the compiler was able to see that a lot of the operations could be changed to bit shitfing which is way faster!

In [11]:
```
// multiplication.c
#include <math.h>
int main(){
    int j = 0;
    for (int i = 0; i < 1000000000; i++){
        j = i*3;
    }
}
```

$ sudo perf stat ./multiplication Performance counter stats for './multiplication': 693.11 msec task-clock # 0.999 CPUs utilized 1 context-switches # 1.443 /sec 0 cpu-migrations # 0.000 /sec 50 page-faults # 72.139 /sec 2,595,447,558 cycles # 3.745 GHz (83.27%) 28,501 stalled-cycles-frontend # 0.00% frontend cycles idle (83.26%) 1,837,508 stalled-cycles-backend # 0.07% backend cycles idle (83.26%) 8,001,252,232 instructions # 3.08 insn per cycle # 0.00 stalled cycles per insn (83.26%) 1,000,344,314 branches # 1.443 G/sec (83.48%) 35,998 branch-misses # 0.00% of all branches (83.46%) 0.694004320 seconds time elapsed 0.694155000 seconds user 0.000000000 seconds sys

In [ ]:
```
// division.c
#include <math.h>
int main(){
    int j = 0;
    for (int i = 0; i < 1000000000; i++){
        j = i/3;
    }
}
```

$ sudo perf stat ./division Performance counter stats for './division': 802.93 msec task-clock # 0.999 CPUs utilized 1 context-switches # 1.245 /sec 0 cpu-migrations # 0.000 /sec 51 page-faults # 63.518 /sec 3,006,455,583 cycles # 3.744 GHz (83.06%) 38,173 stalled-cycles-frontend # 0.00% frontend cycles idle (83.06%) 2,806,425 stalled-cycles-backend # 0.09% backend cycles idle (83.53%) 12,000,690,795 instructions # 3.99 insn per cycle # 0.00 stalled cycles per insn (83.56%) 1,000,527,651 branches # 1.246 G/sec (83.56%) 49,178 branch-misses # 0.00% of all branches (83.23%) 0.803838726 seconds time elapsed 0.804003000 seconds user 0.000000000 seconds sys

In [ ]:
```
// sqrt.c
#include <math.h>
int main(){
    int j = 0;
    for (int i = 0; i < 1000000000; i++){
        j = sqrt(i);
    }
}
```

$ sudo perf stat ./sqrt Performance counter stats for './sqrt': 2,675.32 msec task-clock # 1.000 CPUs utilized 4 context-switches # 1.495 /sec 0 cpu-migrations # 0.000 /sec 61 page-faults # 22.801 /sec 10,017,423,521 cycles # 3.744 GHz (83.26%) 75,012 stalled-cycles-frontend # 0.00% frontend cycles idle (83.25%) 7,201,751 stalled-cycles-backend # 0.07% backend cycles idle (83.38%) 20,004,402,439 instructions # 2.00 insn per cycle # 0.00 stalled cycles per insn (83.40%) 6,001,188,618 branches # 2.243 G/sec (83.40%) 146,444 branch-misses # 0.00% of all branches (83.31%) 2.676314956 seconds time elapsed 2.676377000 seconds user 0.000000000 seconds sys

In [ ]:
```
// sin.c
#include <math.h>
int main(){
    int j = 0;
    for (int i = 0; i < 1000000; i++){
        j = sin(i);
    }
}
```

$ sudo perf stat ./sin Performance counter stats for './sin': 36,694.59 msec task-clock # 1.000 CPUs utilized 81 context-switches # 2.207 /sec 0 cpu-migrations # 0.000 /sec 65 page-faults # 1.771 /sec 137,404,519,046 cycles # 3.745 GHz (83.33%) 1,729,605 stalled-cycles-frontend # 0.00% frontend cycles idle (83.33%) 91,795,142 stalled-cycles-backend # 0.07% backend cycles idle (83.33%) 336,083,010,078 instructions # 2.45 insn per cycle # 0.00 stalled cycles per insn (83.33%) 23,884,801,515 branches # 650.908 M/sec (83.33%) 12,191,316 branch-misses # 0.05% of all branches (83.33%) 36.697755393 seconds time elapsed 36.695391000 seconds user 0.000000000 seconds sys

Part 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....

In [12]:
```
# include <time.h>
# include <math.h>
# include <stdio.h>
# include <stdlib.h>

int main(int argc, char **argv) {
  int i,j;
  int n = 128;
  double sum;
  clock_t end, start;
  double arr[128][128];

// THIS FILLS THE MATRIX WITH NUMBERS
  for (i=0; i<n; i++){
    for (j=0; j<n; j++){
      arr[i][j] = (double) rand()/RAND_MAX;
    }
  }

  sum = 0;

  start = clock();

// ROW MAJOR WORK
// YOU'LL NEED TO TIME IT
for (i = 0; i<n; i++){ // iterate over rows
  for (j = 0; j<n; j++){ // iterate over columns
    sum += arr[i][j];
  }
}
      end = clock();

// NOTE:  YOU'LL NEED TO PROVIDE MEANING TO end AND start
  printf("Row Major: sum = %lf and Clock Ticks are %ld\n",sum,end-start);

//ADD YOUR COLUMN MAJOR WORK
// YOU'LL NEED TO TIME IT

  return 0;
}
```

```
Row Major: sum = 8143.619987 and Clock Ticks are 55
```

$ valgrind --tool=cachegrind ./loopsstaticshell ==2497347== Cachegrind, a cache and branch-prediction profiler ==2497347== Copyright (C) 2002-2017, and GNU GPL'd, by Nicholas Nethercote et al. ==2497347== Using Valgrind-3.18.1 and LibVEX; rerun with -h for copyright info ==2497347== Command: ./loopsstaticshell ==2497347== --2497347-- warning: L3 cache found, using its data for the LL simulation. Row Major: sum = 8143.619987 and Clock Ticks are 877 Column Major: sum = 16287.239974 and Clock Ticks are 808 ==2497347== ==2497347== I refs: 1,862,479 ==2497347== I1 misses: 1,484 ==2497347== LLi misses: 1,457 ==2497347== I1 miss rate: 0.08% ==2497347== LLi miss rate: 0.08% ==2497347== ==2497347== D refs: 840,028 (661,300 rd + 178,728 wr) ==2497347== D1 misses: 22,919 (20,193 rd + 2,726 wr) ==2497347== LLd misses: 3,906 (1,376 rd + 2,530 wr) ==2497347== D1 miss rate:

2.7% (3.1% + 1.5%) ==2497347== LLd miss rate: 0.5% (0.2% + 1.4%) ==2497347== ==2497347== LL refs: 24,403 (21,677 rd + 2,726 wr) ==2497347== LL misses: 5,363 (2,833 rd + 2,530 wr) ==2497347== LL miss rate: 0.2% (0.1% + 1.4%)

In [14]:
```
# include <time.h>
# include <math.h>
# include <stdio.h>
# include <stdlib.h>

int main(int argc, char **argv) {
  int i,j;
  int n = 128;
  double sum;
  clock_t end, start;
  double *arr = malloc(n*n*sizeof(double));

// THIS FILLS THE MATRIX WITH NUMBERS
  for (i=0; i<n; i++){
    for (j=0; j<n; j++){
      arr[i*n+j] = (double) rand()/RAND_MAX;
    }
  }

  sum = 0;

  start = clock();
// ROW MAJOR WORK
// YOU'LL NEED TO TIME IT
for (i = 0; i<n; i++){ // iterate over rows
  for (j = 0; j<n; j++){ // iterate over columns
    sum += arr[i*n + j];
  }
}
end = clock();

  printf("Row Major: sum = %lf and Clock Ticks are %ld\n",sum,end-start);

//ADD YOUR COLUMN MAJOR WORK
// YOU'LL NEED TO TIME IT

  return 0;
}
```

```
Row Major: sum = 8143.619987 and Clock Ticks are 54
```

$ valgrind --tool=cachegrind ./loopsdynamicshell ==2497352== Cachegrind, a cache and branch-prediction profiler ==2497352== Copyright (C) 2002-2017, and GNU GPL'd, by Nicholas Nethercote et al. ==2497352== Using Valgrind-3.18.1 and LibVEX; rerun with -h for copyright info ==2497352== Command: ./loopsdynamicshell ==2497352== --2497352-- warning: L3 cache found, using its data for the LL simulation. Row Major: sum = 8143.619987 and Clock Ticks are 976 Column Major: sum = 16287.239974 and Clock Ticks are 903 ==2497352== ==2497352== I refs: 2,009,802 ==2497352== I1 misses: 1,483 ==2497352== LLi misses: 1,453 ==2497352== I1 miss rate: 0.07% ==2497352== LLi miss rate: 0.07% ==2497352== ==2497352== D refs: 938,290 (759,538 rd + 178,752 wr) ==2497352== D1 misses: 22,889 (20,161 rd + 2,728 wr) ==2497352== LLd misses: 3,975 (1,338 rd + 2,637 wr) ==2497352== D1 miss rate: 2.4% (2.7% + 1.5%) ==2497352== LLd miss rate: 0.4% (0.2% + 1.5%) ==2497352== ==2497352== LL refs: 24,372 (21,644 rd + 2,728 wr) ==2497352== LL misses: 5,428 (2,791 rd + 2,637 wr) ==2497352== LL miss rate: 0.2% (0.1% + 1.5%)

Running with multiple and graphing

./static_vs_dynamic.png Here we can clearly see there is a large discrepancy in row major vs column major addition!

Part 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.

In [ ]:

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

// THis makes the code more reusable, we can have a bunch of functions and s
typedef void (*TranspositionFunc)(char *);

void reverse_string(char *str) {
    if (str == NULL) return;

    char *start = str;
    char *end = str + strlen(str) - 1;

    char temp;

    while (start < end) {
        temp = *start;
        *start = *end;
        *end = temp;

        start++;
        end--;
    }
}

// Accepts the data (str) and the function to apply to it (func)
void apply_and_print(char *str, TranspositionFunc func) {
    printf("\n--- Transposition Report ---\n");
    printf("Original String:   %s\n", str);

    func(str);

    printf("Transformed String: %s\n", str);
    printf("--------------------------\n");
}

int main() {
```

```
    size_t buffer_size = 1024;

    // Dynamic Allocation: Allocate memory on the heap for the input
    char *input_buffer = (char *)malloc(buffer_size * sizeof(char));

    if (input_buffer == NULL) {
        fprintf(stderr, "Memory allocation failed\n");
        return 1;
    }

    printf("Enter a string to transpose: ");

    // Read input from stdin
    if (getline(&input_buffer, &buffer_size, stdin) != -1) {
        // Strip the newline character
        char *newline = strchr(input_buffer, '\n');
        if (newline) *newline = '\0';

        // Pass in the string and the function reverse_string
        apply_and_print(input_buffer, reverse_string);
    } else {
        printf("Error reading input.\n");
    }

    // Clean up heap memory
    free(input_buffer);

    return 0;
}
```

$ ./part6 Enter a string to transpose: Hello World! -Trevor --- Transposition Report --- Original String: Hello World! -Trevor Transformed String: roverT- !dlroW olleH -------------------------

Final Questions

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 fgets to grab input from the terminal. This is how I learned to do it, but I can see how it is less parallelizable than grabbing from a file or using clargs!

b. How did you manage or allocate the strings? (Static or dynamic)

> I used static allocation, I wasn't quite as familiar with dynamic string allocation in C until midway through this assignment!

2. For Archimedes algorithm

a. How did you time your program?

> I timed it using `perf stat` I did it this way to keep overhead in my executable lower!

b. Were there any issues with precision and/or convergence that you noticed?

> There were issues if I ran the code past the suggested 100 limit but, precision up until that point was fine, there was probably some floating point arithmetic errors but, nothing that made my answers too far off!

3. For Matrix-vector multiplication

a. How did you allocate and access your matrix?

> I allocated it and accessed it dynamically by taking in file information.

b. Were there any challenges in reading in the file?

> It took me a minute to stop and think aboout the best way to read the file in, but fscanf was pretty elegant!

c. Was there anything special about the actual computation?

> Not in particular, it took looking back at linear algebra notes and using an online matrix calculator to sanity check my results before I was comfortable submitting.

d. What was your strategy for timing?

> Again to reduce overhead in my program, I utilized `perf stat` which was able to give very detailed and helpful timing information

4. For measuring the speed of arithmetic computations

a. What was your timing strategy?

> Again to reduce overhead in my program, I utilized `perf stat` which was able to give very detailed and helpful timing information

b. Are all arithmetic operations created equal?

> No not all arithmetic operations are equal! The compiler can try and do some optimizations like if it sees a div with a multiple of 2 it can change that to a left shift but, the operations without dedicated operators are much slower! Particularly sin! A way to combat the slowness of something like sin is to use pre-calculated tables or values!

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?

> We had to be much more explicit in the static arrays, changing the array size and n, there were some small performance increases with the dynamic but that was just in the startup phase, once we got into the loop, the arrays were identical.

b. What did you observe about differences in program behavior in row-major vs. column major computations and how do you explain it?

> Row-major order ended up taking over in the speed game, likely due to the cache's spatial locality! It is much easier to access memory close to what you just accessed than memory further away! We had much fewer cache misses with row major computations!